

# Data Compression with T-Codes

Christian Müller, cm285@web.de

Rolf Schimpfky, rs8086@web.de

The University of Auckland

## Abstract

The ongoing process of research on T-Codes demanded further investigation of the compression opportunities of the concept. The introduction of T-prefix matrices simplified the project noticeably. The Calgary Corpus was used as benchmark to make the results comparable to other compression algorithms. The different approaches achieved good results, which were discussed and compared with GZIP 1.3, BZIP 1.0.2, and compress 4.2.4.

## 1 Introduction

The concept of T-Codes was developed by Mark Titchener about two decades ago[8][9]. T-Codes have continuously been researched at The University of Auckland since. Radu Nicolescu and Ulrich Günther also published several papers on the topic, some of which formed the basis of our project[5][4][6]. The concept works similar to other coding algorithms, e.g., LempelZiv. The question hence arose whether T-Codes may be used for compression purposes as well. We will therefore introduce T-prefix matrices, which will complete our introductory section on T-Codes. We will then show some of their characteristics on the basis of the Calgary Corpus[1]. In section four, we will introduce different approaches how to save a T-prefix matrix conveniently using the gained knowledge. We will finally compare and discuss the compression results.

## 2 An Introduction to T-Codes

### 2.1 T-Augmentation and T-Code Sets

We consider a set of symbols as a finite alphabet  $S$  with a size of  $\# S$ , which is the number of symbols in the alphabet[5]. The alphabet may be illustrated as a tree of depth one in which each symbol is represented by a leaf node. We obtain a set of codewords of length one. They are represented by the tree's leaf nodes. In order to expand the length of the possible codewords, we may append one or more copies of the tree to one of the leaf nodes. The selected leaf node becomes an internal node as it connects the original tree with its copies. The expanded codeword set is represented by the new leaf nodes. The length of the longest possible codeword has also increased. We may repeat this procedure by making use of the newly created tree. That example considering a tree shows the concept of T-augmentation, which is defined as follows[5]:

#### Definition 2.1 (T-Augmentation)

The mapping

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

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

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

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.

We may continue with T-Code sets and intermediate T-Code sets both of which are defined with the help of T-augmentation in [5]:

#### Definition 2.2 (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  $p_1, p_2, \dots, p_n$  and T-expansion parameters  $k_1, k_2, \dots, k_n$  respectively, such that

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

is called a **T-Code set at T-augmentation level  $n$** . We write

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

**Definition 2.3 (Intermediate T-Code Sets)**

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

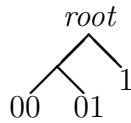
The following example illustrates how T-Code sets may be obtained through T-augmentation:

**Example 2.4 (T-Augmentation)**



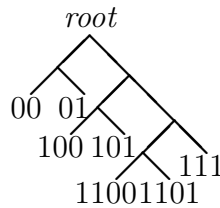
Consider the alphabet  $S = \{0, 1\}$ , which is also the T-Code set at T-augmentation level 0. We choose  $p_1 = 0$  and  $k_1 = 1$  for the first T-augmentation level. As the result, the T-Code set at T-augmentation level 1 is

$$S_{(0)}^{(1)} = \{1, 00, 01\}.$$



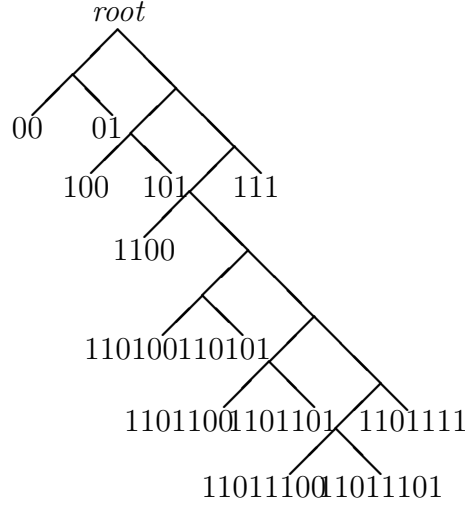
Next, we set  $p_2 = 1$  and  $k_2 = 2$ . Thus, the T-Code set at T-augmentation level 2 is

$$S_{(0,1)}^{(1,2)} = \{00, 01, 100, 101, 111, 1100, 1101\}.$$



Finally, we assume  $p_3 = 1101$  and  $k_3 = 1$ . As the result, we obtain the following T-Code set:

$$S_{(0,1,1101)}^{(1,2,1)} = \{00, 01, 100, 101, 111, 1100, 110100, 110101, 1101100, 1101101, 1101111, 11011100, 11011101\}$$



We shall mention two important properties of T-Code sets. Any given T-Code set is both prefix-free and complete[5]. As we may have noticed as well, there are  $n$  longest codewords in a T-Code set based on an alphabet with  $n$  symbols[4].

## 2.2 String Decomposition

The longest codewords differ only in their last character, which is called the "literal symbol". In addition, they contain all T-prefixes and T-expansion parameters of a given T-Code set in the following way[4]:

$$s_c = p_n^{k_n} p_{n-1}^{k_{n-1}} \dots p_1^{k_1} c$$

According to R. Nicolescu[6], the T-Code set  $S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)}$  is properly defined by the T-prefix part of the longest codewords,  $p_n^{k_n} p_{n-1}^{k_{n-1}} \dots p_1^{k_1}$ . U. Günther has formulated two theorems for the decomposition of T-Code sets[5]:

### Theorem 2.5 (Decomposition of T-Code Codewords)

For all codewords  $x \in S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)}$ , a decomposition

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1} k'_0, \quad (2)$$

exists such that  $0 \leq k'_i \leq k_i$  for  $i = 0, 1, \dots, n$  and  $k_0 = \#S - 1$ . This decomposition of  $x$  is unique, i.e., there exists exactly one set of  $k'_i$  for which equation (2) is satisfied.

**Theorem 2.6 (Decomposition for Intermediate T-Code Sets)**

For any  $m \leq n$ , the decomposition of  $x \in S_{(p_1, p_2, \dots, p_m)}^{(k_1, k_2, \dots, k_m)}$  as

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1} k'_0 \quad (3)$$

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

Any decomposition of T-Code codewords may be splitted into different parts which are called as follows[5]:

**Definition 2.7 (T-Expansion Indices and Literal Symbol)**

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

**Algorithm of Decomposition**

M. Titchener describes the original algorithm of decomposition that benefits from this relation between a T-Code set and its longest codewords. We may paraphrase the algorithm of decomposition by U. Günther[4]:

1. start at T-augmentation level 0 with the set S.
2. decode  $s_c$  over the set at the current T-augmentation level, starting on the left side of the string
3. if  $s_c$  decodes as a single codeword,  $s_c$  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  $s_c$ . 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.

Note, it is not necessary to know the T-augmentation level of the T-Code set that contains the given longest codeword. The following example may illustrate the algorithm.

**Example 2.8 (Decomposition of One Longest Codeword)**

Consider the alphabet  $S = \{0, 1\}$  and the string  $s_c = 1101110$  to be one of the longest codewords. In the first step, we use the alphabet to decode the string (with each dot indicating the ending of a codeword):

1.1.0.1.1.1.0

The first T-prefix is 1 since it is the second codeword from the right. The associated T-expansion parameter is 3 as the T-prefix appears three times in a row. The resulting T-Code set at T-augmentation level 1 is

$$S_{(1)}^{(3)} = \{0, 10, 110, 1110, 1111\}.$$

Next, we analyse the string regarding this T-Code set.

110.1110

The second T-prefix is 110. It appears once, hence the T-expansion parameter is 1. The final T-Code set is

$$S_{(1,110)}^{(3,1)} = \{1, 10, 1100, 1110, 1111, 11010, 110110, 1101110, 1101111\}.$$

However, it is difficult to find out what the original string was that led to the present T-prefixes and T-expansion parameters. We may see in the example above that we do not know which of the longest codewords in the final T-Code set was the original string. We thus assume the original string to be mere the T-prefix part and append an ambiguous letter which is an element of the alphabet<sup>1</sup>. We then select one of the longest codewords of the final T-Code set we got after decomposing the string. We cut off the last symbol of the chosen codeword and reobtain the original string. It does not matter which of the longest codewords we pick out. We know that the longest codewords differ only in the last character, which is the one to be chopped off. As a result, it is possible to derive an encoding and decoding technique from T-decomposition. If we consider a file as a string of characters, we may convert it into a list of T-prefixes and a list of T-expansion parameters. Later on, we will find out whether it is possible to store these lists more efficiently than the original file.

---

<sup>1</sup>In accordance with the algorithm by U. Günther[5]

**Example 2.9 (String Decomposition)**

Consider the string  $s_c = 1101110$  and the alphabet  $S = \{0, 1\}$ . In the first step, we add the ambiguous character  $\beta \in S$  to the end of the string and analyze the string according to the alphabet:

$$1.1.0.1.1.1.0.\beta$$

At  $T$ -augmentation level one, the  $T$ -prefix is 0 and the  $T$ -expansion parameter is 1 as its neighbour to the left is 1. The corresponding  $T$ -Code set is

$$S_{(0)}^{(1)} = \{1, 00, 01\}.$$

We go on to the next  $T$ -augmentation level:

$$1.1.01.1.1.0\beta$$

The new  $T$ -prefix is 1 and the  $T$ -expansion parameter is 2 because it appears twice in row. Hence, the new  $T$ -Code set is

$$S_{(0,1)}^{(1,2)} = \{00, 01, 100, 101, 111, 1100, 1101\}.$$

We again examine the string according to the latter  $T$ -Code set:

$$1101.110\beta$$

We get the last  $T$ -prefix which is 1101 and appears once. The final  $T$ -Code set is

$$S_{(0,1,1101)}^{(1,2,1)} = \{00, 01, 100, 101, 111, 1100, 110100, 110101, 1101100, 1101101, 1101111, 11011100, 11011101\}.$$

Finally, we choose one of the longest codewords, chop off the last character, and reobtain the original string 1101110.

J. Yang and U. Günther[11] recently implemented a fast version of this algorithm, which we used as well. The part of the program we used returned the  $T$ -prefixes and the  $T$ -expansion parameters of the input string. For the recovery of the original string from the  $T$ -prefixes and  $T$ -expansion parameters, we implemented an algorithm based on Theorem 2.6:

```
string =  $\emptyset$ 
for  $i = width$  to 1 do
```

```

    string ← string + ki · prefix[i]
  end for
  echo string

```

Starting with the empty word  $w = \emptyset$ , we concatenate each prefix  $p_i$   $k_i$ -times to  $w$  from the right-hand side. We begin with the prefix of the greatest T-augmentation level  $i = \text{width}$  and step down to level  $i = 1$  in order to obtain the original string.

### 2.3 T-Depletion Codes

By now, we are able to decompose any codeword of a given T-Code set in a unique way. In order to store the T-expansion indices and literal symbols conveniently, we introduce U. Günther's definition of T-depletion codewords as multibase numbers[5]:

#### Definition 2.10 (Multibase Numbers)

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

$$0 \leq k'_i \leq k_i. \quad (4)$$

#### Definition 2.11 (T-Depletion Codewords)

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

$$x = p_n^{k'_n} p_{n-1}^{k'_{n-1}} \dots p_1^{k'_1} k'_0. \quad (5)$$

#### Example 2.12 (T-Depletion Codewords)

Consider the T-Code set  $S_{(0,1,1101)}^{(1,2,1)}$ . We find the appropriate T-depletion codeword for each element of the T-Code set in Table 1.

### 2.4 T-Prefix Matrix

We start by introducing the following notation:

$$S_f = \begin{cases} S & \text{for } f = 0 \\ S_{(p_1, p_2, \dots, p_f)}^{(k_1, k_2, \dots, k_f)} & \text{for } f \geq 1. \end{cases} \quad (6)$$

T-Code	structure	T-depletion codeword
		$(k'_3, k'_2, k'_1, k'_0)$
00	$(1101)^0(1)^0(0)^10$	$(0, 0, 1, 0)$
01	$(1101)^0(1)^0(0)^11$	$(0, 0, 1, 1)$
100	$(1101)^0(1)^1(0)^10$	$(0, 1, 1, 0)$
101	$(1101)^0(1)^1(0)^11$	$(0, 1, 1, 1)$
111	$(1101)^0(1)^2(0)^01$	$(0, 2, 0, 1)$
1100	$(1101)^0(1)^2(0)^10$	$(0, 2, 1, 0)$
110100	$(1101)^1(1)^0(0)^10$	$(1, 0, 1, 0)$
110101	$(1101)^1(1)^0(0)^11$	$(1, 0, 1, 1)$
1101100	$(1101)^1(1)^1(0)^10$	$(1, 1, 1, 0)$
1101101	$(1101)^1(1)^1(0)^11$	$(1, 1, 1, 1)$
1101111	$(1101)^1(1)^2(0)^01$	$(1, 2, 0, 1)$
11011100	$(1101)^1(1)^2(0)^10$	$(1, 2, 1, 0)$
11011101	$(1101)^1(1)^2(0)^11$	$(1, 2, 1, 1)$

Table 1: T-Codes and their corresponding T-depletion codewords

Each T-prefix  $p_g$  of the final T-Code set  $S_n$  is one of the codewords of the intermediate T-Code set  $S_{g-1}$  at the previous T-augmentation level  $g-1$ , as we may derive from definition 2.1 (T-Augmentation) and definition 2.2 (T-Code Sets). According to theorem 2.5 (Decomposition of T-Code Codewords), we may decompose  $p_g \in S_{g-1}$  as follows:

$$p_g = p_{g-1}^{k'_{g-1}} p_{g-2}^{k'_{g-2}} \dots p_1^{k'_1} k'_0 \text{ for } 1 \leq g \leq n. \quad (7)$$

That leads us to the T-depletion codeword for an arbitrary T-prefix  $p_g$  of the T-Code set  $S_n$ :

$$(k'_{g-1}, k'_{g-2}, \dots, k'_1, k'_0) \quad (8)$$

Ulrich Speidel[7] has suggested to record these codewords in a matrix. Such a matrix presents both the T-expansion indices and the literal symbol for each T-prefix in the following way:

$$\begin{array}{c|cccccc}
- & k'_0 & k'_1 & \cdots & k'_{n-2} & k'_{n-1} \\
p_1 & k'_{1,0} & k'_{1,1} & \cdots & k'_{1,n-2} & k'_{1,n-1} \\
p_2 & k'_{2,0} & k'_{2,1} & \cdots & k'_{2,n-2} & k'_{2,n-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
p_{n-1} & k'_{n-1,0} & k'_{n-1,1} & \cdots & k'_{n-1,n-2} & k'_{n-1,n-1} \\
p_n & k'_{n,0} & k'_{n,1} & \cdots & k'_{n,n-2} & k'_{n,n-1}
\end{array} \quad (9)$$

As the result, we introduce the following definition:

**Definition 2.13 (T-Prefix Matrix)**

Let  $S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)}$  be a T-Code set with the T-prefixes  $p_g = p_{g-1}^{k'_{g,g-1}} p_{g-2}^{k'_{g,g-2}} \dots p_1^{k'_{g,1}} k'_{g,0}$ ,  $1 \leq g \leq n$ . The corresponding **T-prefix matrix** consists of the entries  $k'_{g,i}$  with  $0 \leq i \leq n-1$ , and  $k'_{g,i} = 0$  for  $i \geq g$ .

The following passages describe the algorithms we used to convert the list of T-prefixes of a given T-Code set into the corresponding T-prefix matrix and vice versa.

**From a List of T-Prefixes to the T-Prefix Matrix**

We start by creating a matrix from a given list of T-prefixes. The number of prefixes is saved separately as *width*. Furthermore, we create an empty square matrix of size *width*. By now, we may investigate for each T-prefix whether it is also a prefix of another T-prefix at a higher T-augmentation level. We begin at T-augmentation level *width* and continue until level 1. If we find a T-prefix to be a prefix of another one, we count the occurrences of it in the other T-prefix. We store the obtained number at the appropriate place in the matrix. Finally, the literal symbol of each investigated T-prefix is saved in the matrix. The elements of the matrix not placed through the process of storing are set zero. The exact algorithm may be described by the following pseudo-code:

```

width ← number of T-prefixes
matrix  $M_{width \times width}$ 
for all T-prefixes  $p$  with row  $g = width$  to 1 do
  for all T-prefixes  $q$  with row  $f = width$  to  $g + 1$  do
    if  $p$  is a prefix of  $q$  then
       $k' \leftarrow$  number of occurrences in a row of  $p$  as a prefix of  $q$ 
       $q \leftarrow$  delete the prefix  $p$   $k'$  times from the beginning of  $q$ 
       $M_{(f,g)} = k'$ 
    end if
  end for
   $literal \leftarrow$  last character of  $p$ 
   $p \leftarrow p$  without its last character
   $M_{(g,0)} \leftarrow literal$ 
end for

```

**Example 2.14 (Conversion from T-Prefixes into a T-Prefix Matrix)**

Consider the T-Code set  $S_{(1,10,0,001010,00101011)}^{(1,1,2,1,1)}$ . We find the appropriate T-depletion codewords for the T-prefixes in the following table:

$g$	$T$ -prefix	structure	$T$ -depletion codeword
			$(k'_{g,g-1}, k'_{g,g-2}, \dots, k'_{g,0})$
1	1	0	(1)
2	10	$(1)^1 0$	(1, 0)
3	0	$(10)^0 (1)^0 0$	(0, 0, 0)
4	001010	$(0)^2 (10)^1 (1)^1 0$	(2, 1, 1, 0)
5	00101011	$(001010)^1 (0)^0 (10)^0 (1)^1 1$	(1, 0, 0, 1, 1)

The rest of the entries have to be set zero, according to definition 2.13 ( $T$ -Prefix Matrix):  $k'_{1,1} = k'_{1,2} = k'_{1,3} = k'_{1,4} = k'_{2,2} = k'_{2,3} = k'_{2,4} = k'_{3,3} = k'_{3,4} = k'_{4,4} = 0$ . We obtain the following  $T$ -prefix matrix:

$$\begin{pmatrix} k'_{1,0} & k'_{1,1} & k'_{1,2} & k'_{1,3} & k'_{1,4} \\ k'_{2,0} & k'_{2,1} & k'_{2,2} & k'_{2,3} & k'_{2,4} \\ k'_{3,0} & k'_{3,1} & k'_{3,2} & k'_{3,3} & k'_{3,4} \\ k'_{4,0} & k'_{4,1} & k'_{4,2} & k'_{4,3} & k'_{4,4} \\ k'_{5,0} & k'_{5,1} & k'_{5,2} & k'_{5,3} & k'_{5,4} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

## From a $T$ -Prefix Matrix to the $T$ -Prefix List

In order to decode a  $T$ -prefix matrix, we process each row  $g$  of the matrix separately. We convert it back into a  $T$ -prefix  $p_g$ , beginning with the first row and iterating to the last one. When processing a row  $g$ , we start with  $p_g = k'_{g,0}$ , which is the literal character. We define the index  $i$  that moves over the current row, starting at 1. For each  $i$ , we add  $p_{g,i}^{k'_{g,i}}$  to  $p_g$ .

```

for row = 1 to width do
  word  $\leftarrow M_{(row,0)}$ 
  for col = 1 to row - 1 do
    n  $\leftarrow M_{(row,col)}$ 
    word  $\leftarrow n \cdot prefix[col] + word$ 
  end for
  prefix[row]  $\leftarrow word$ 
end for

```

## 2.5 Contiguous Range Indices

We may regard a  $T$ -Code set as a tree again. If we have a look at definition 2.11 ( $T$ -Depletion Codewords), we may notice that  $T$ -depletion codes

describe not only the leaf nodes, which are the codewords of the T-Code set, but also the internal nodes, which are not part of the T-Code set. Therefore, the created model still contains redundant information. In contrast, the contiguous range index conversion is a unique enumeration scheme which numbers consecutively the codewords of a T-Code set. The conversion assigns positive integers only to the tree's external nodes. We may adapt two propositions by U. Günther[5] to Definition 2.13 (T-Prefix Matrix). The first one deals with the trivial case.

**Proposition 2.15**

The mapping  $I_0 : S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)} \rightarrow \mathbb{N}$ , with  $i = 1, \dots, n$  and

$$I_0(p_i) = k'_{i,0}, \quad (10)$$

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

Note: The literal symbols  $k'_{i,0}$  used in Proposition 2.15 have to be natural numbers. Therefore, it is necessary that the elements of the alphabet are numbered from 0 up to  $\#S - 1$ .

The second proposition describes how we may derive contiguous integer indices at higher T-augmentation levels. To understand it correctly, we need to consider the cardinality of a T-Code set[5] basing on the number of elements in the alphabet  $\#S$ .

**Theorem 2.16**

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

$$\#S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)} = 1 + (\#S - 1) \prod_{i=1}^n (k_i + 1). \quad (11)$$

We may use the following notation for the cardinality of a T-Code set:

$$\#S_f = \begin{cases} \#S & \text{for } f = 0 \\ \#S_{(p_1, p_2, \dots, p_f)}^{(k_1, k_2, \dots, k_f)} & \text{for } f \geq 1. \end{cases} \quad (12)$$

Additionally, another condition[5] for the cardinality may be used:

$$\#S_i = k_i(\#S_{i-1} - 1) + \#S_{i-1}, \text{ with } i \geq 1. \quad (13)$$

**Proposition 2.17**

Let  $I_m(p_i)$  be the contiguous integer index at T-augmentation level  $m$  with

$0 \leq m < n$  and  $i = 1, \dots, n$ . The mapping  $I_{m+1} : S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)} \rightarrow \mathbb{N}$ , with

$$I_{m+1}(p_i) = \begin{cases} I_m(p_i) + k'_{i,m+1} (\#S_m - 1) \text{ for} \\ \quad I_m(p_i) \leq I_m(p_{m+1}) \vee k'_{i,m+1} = k_{m+1} \\ I_m(p_i) + k'_{i,m+1} (\#S_m - 1) - 1 \text{ for} \\ \quad I_m(p_i) > I_m(p_{m+1}) \wedge k'_{i,m+1} < k_{m+1} \end{cases} \quad (14)$$

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

We are now able to convert each row of a T-prefix matrix into a contiguous range index. Consequently, a T-prefix matrix may be mapped onto a vector of contiguous range indices:

$$\begin{pmatrix} I_0(p_1) \\ I_1(p_2) \\ \vdots \\ I_{n-1}(p_n) \end{pmatrix} \quad (15)$$

The conversion is a one-to-one mapping such that we may reobtain the associated T-prefix matrix of a given vector of contiguous range indices. The following formulae[5], which we adapted to definition 2.13 (T-Prefix Matrix), describe how we may acquire the corresponding T-depletion codeword from a given index  $I_n(p_i)$  for a T-prefix  $p_i$  of the T-Code set  $S_{(p_1, p_2, \dots, p_n)}^{(k_1, k_2, \dots, k_n)}$ .

$$k'_{i,m} = \begin{cases} \left\lfloor \frac{I_m(p_i)}{\#S_{m-1}-1} \right\rfloor \text{ for } \left\lfloor \frac{I_m(p_i)}{\#S_{m-1}-1} \right\rfloor \leq k_m \\ k_m \text{ otherwise} \end{cases} \quad (16)$$

$$I_{m-1}(p_i) = \begin{cases} I_m(p_i) - k'_{i,m} (\#S_{m-1} - 1) \text{ for} \\ \quad I_m(p_i) - k'_{i,m} (\#S_{m-1} - 1) < I_{m-1}(p_m) \vee k_m = k'_{i,m} \\ I_m(p_i) - k'_{i,m} (\#S_{m-1} - 1) + 1 \text{ otherwise} \end{cases} \quad (17)$$

### Example 2.18 (From a T-Prefix Matrix to Contiguous Range Indices)

We consider the string  $s_c = 0010101100101000101\beta$  with  $\beta \in \{0, 1\}$ , which

is one of the longest codewords of the T-Code set  $S_{(1,10,0,001010,00101011)}^{(1,1,2,1,1)}$ . We have derived the corresponding T-prefix matrix in Example 2.14:

$$\begin{pmatrix} k'_{1,0} & k'_{1,1} & k'_{1,2} & k'_{1,3} & k'_{1,4} \\ k'_{2,0} & k'_{2,1} & k'_{2,2} & k'_{2,3} & k'_{2,4} \\ k'_{3,0} & k'_{3,1} & k'_{3,2} & k'_{3,3} & k'_{3,4} \\ k'_{4,0} & k'_{4,1} & k'_{4,2} & k'_{4,3} & k'_{4,4} \\ k'_{5,0} & k'_{5,1} & k'_{5,2} & k'_{5,3} & k'_{5,4} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Next, we determine the cardinality of each intermediate T-Code set:

$$\begin{aligned} \#S_0 &= 2, \\ \#S_1 &= 3, \\ \#S_2 &= 5, \\ \#S_3 &= 13, \\ \#S_4 &= 25, \\ \#S_5 &= 49. \end{aligned}$$

We continue with the index conversion:

$m$	$i$	$I_m(p_i) \leq I_m(p_{m+1})$ , if necessary $k'_{i,m+1} = k_{m+1}$	$I_{m+1}(p_i)$
–	1	–	$I_0(p_1) = k'_{1,0} = 1$
–	2	–	$I_0(p_2) = k'_{2,0} = 0$
–	3	–	$I_0(p_3) = k'_{3,0} = 0$
–	4	–	$I_0(p_4) = k'_{4,0} = 0$
–	5	–	$I_0(p_5) = k'_{5,0} = 1$
0	2	$I_0(p_2) = 0 \leq 1 = I_0(p_1)$ is true	$I_1(p_2) = I_0(p_2) + k'_{2,1}(\#S_0 - 1) = 1$
0	3	$I_0(p_3) = 0 \leq 1 = I_0(p_1)$ is true	$I_1(p_3) = I_0(p_3) + k'_{3,1}(\#S_0 - 1) = 0$
0	4	$I_0(p_4) = 0 \leq 1 = I_0(p_1)$ is true	$I_1(p_4) = I_0(p_4) + k'_{4,1}(\#S_0 - 1) = 1$
0	5	$I_0(p_5) = 1 \leq 1 = I_0(p_1)$ is true	$I_1(p_5) = I_0(p_5) + k'_{5,1}(\#S_0 - 1) = 2$
1	3	$I_1(p_3) = 0 \leq 1 = I_1(p_2)$ is true	$I_2(p_3) = I_1(p_3) + k'_{3,2}(\#S_1 - 1) = 0$
1	4	$I_1(p_4) = 1 \leq 1 = I_1(p_2)$ is true	$I_2(p_4) = I_1(p_4) + k'_{4,2}(\#S_1 - 1) = 3$
1	5	$I_1(p_5) = 2 \leq 1 = I_1(p_2)$ is false, $k'_{5,2} = 0 = 1 = k_2$ is false	$I_2(p_5) = I_1(p_5) + k'_{5,2}(\#S_1 - 1) - 1 = 1$
2	4	$I_2(p_4) = 3 \leq 0 = I_2(p_3)$ is false, $k'_{4,3} = 2 = 2 = k_3$ is true	$I_3(p_4) = I_2(p_4) + k'_{4,3}(\#S_2 - 1) = 11$
2	5	$I_2(p_5) = 1 \leq 0 = I_2(p_3)$ is false, $k'_{5,3} = 0 = 2 = k_3$ is false	$I_3(p_5) = I_2(p_5) + k'_{5,3}(\#S_2 - 1) - 1 = 0$
3	5	$I_3(p_5) = 0 \leq 11 = I_3(p_4)$ is true	$I_4(p_5) = I_3(p_5) + k'_{5,4}(\#S_3 - 1) = 12$

Eventually, we obtain the following vector of contiguous range indices:

$$\begin{pmatrix} 1 \\ 1 \\ 0 \\ 11 \\ 12 \end{pmatrix}$$

**Example 2.19 (From Contiguous Range Indices to a T-Prefix Matrix)**

Consider the following values. The alphabet of the underlying T-Code set is  $S = \{0, 1\}$ . The given T-expansion parameters are  $(k_1, k_2, k_3, k_4, k_5) = (1, 1, 2, 1, 1)$ . The corresponding vector of contiguous range indices is

$$(I_0(p_1), I_1(p_2), I_2(p_3), I_3(p_4), I_4(p_5)) = (1, 1, 0, 11, 12).$$

We look for the T-prefix matrix of a final T-Code set at T-augmentation level 5, accordingly. We derive the cardinality of each intermediate T-Code set using theorem 2.16 or formula 13:

$$\#S_0 = 2, \#S_1 = 3, \#S_2 = 5, \#S_3 = 13, \#S_4 = 25, \#S_5 = 49.$$

During the calculation, we use three conditions derived from equations 16 and 17.

Condition one:  $\left\lfloor \frac{I_m(p_i)}{\#S_{m-1}-1} \right\rfloor \leq k_m$

Condition two:  $I_m(p_i) - k'_{i,m}(\#S_{m-1} - 1) < I_{m-1}(p_m)$

Condition three:  $k_m = k'_{i,m}$

$m$	$i$	<i>condition one</i>	<i>condition two</i>	<i>condition three</i>	<i>result</i>
4	5	$1 \leq 1$ is true	–	–	$k'_{5,4} = 1$
		–	$0 < 11$ is true	–	$I_3(p_5) = 0$
3	5	$0 \leq 2$ is true	–	–	$k'_{5,3} = 0$
		–	$0 < 0$ is false	$2 = 0$ is false	$I_2(p_5) = 1$
3	4	$2 \leq 2$ is true	–	–	$k'_{4,3} = 2$
		–	$3 < 0$ is false	$2 = 2$ is true	$I_2(p_4) = 3$
2	5	$0 \leq 1$ is true	–	–	$k'_{5,2} = 0$
		–	$1 < 1$ is false	$1 = 0$ is false	$I_1(p_5) = 2$
2	4	$1 \leq 1$ is true	–	–	$k'_{4,2} = 1$
		–	$1 < 1$ is false	$1 = 1$ is true	$I_1(p_4) = 1$
2	3	$0 \leq 1$ is true	–	–	$k'_{3,2} = 0$
		–	$0 < 1$ is true	–	$I_1(p_3) = 0$
1	5	$2 \leq 1$ is false	–	–	$k'_{5,1} = 1$
		–	$1 < 1$ is false	$1 = 1$ is true	$I_0(p_5) = 1$
1	4	$1 \leq 1$ is true	–	–	$k'_{4,1} = 1$
		–	$0 < 1$ is true	–	$I_0(p_4) = 0$
1	3	$0 \leq 1$ is true	–	–	$k'_{3,1} = 0$
		–	$0 < 1$ is true	–	$I_0(p_3) = 0$
1	2	$1 \leq 1$ is true	–	–	$k'_{2,1} = 1$
		–	$0 < 1$ is true	–	$I_0(p_2) = 0$

We finally obtain the literal characters by using proposition 2.15:

$$k'_{5,0} = 1, k'_{4,0} = 0, k'_{3,0} = 0, k'_{2,0} = 0, k'_{1,0} = 1.$$

The T-prefix matrix is:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 2 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

### 3 Characteristics of T-Prefix Matrices

#### 3.1 Benchmark Calgary Corpus

We will introduce and discuss several approaches of lossless data compression using T-Codes in section 4. In order to be able to compare the different approaches both to each other and to commonly used file compressors like GZIP, we need a benchmark. We have chosen the Calgary Corpus[1] as it is

<i>file name</i>	<i>GZIP 1.3</i>	<i>BZIP 1.0.2</i>	<i>compress 4.2.4</i>
bib	0.315	0.247	0.418
book1	0.408	0.303	0.413
book2	0.338	0.258	0.411
geo	0.669	0.556	0.760
news	0.384	0.315	0.487
obj1	0.480	0.501	0.653
obj2	0.331	0.310	0.521
paper1	0.350	0.311	0.472
paper2	0.362	0.305	0.440
paper3	0.389	0.340	0.476
paper4	0.417	0.390	0.524
paper5	0.418	0.405	0.550
paper6	0.347	0.323	0.491
pic	0.110	0.097	0.121
progc	0.335	0.317	0.483
progl	0.227	0.217	0.379
progp	0.228	0.217	0.389
trans	0.203	0.191	0.408

Table 2: Benchmark results for the file compressors GZIP 1.3, BZIP 1.0.2, and compress 4.2.4. The used files form the Calgary Corpus. All values in bits per bit.

widely used. It contains a mixture of many typical files, e.g., program sources, images, documents, and binary files. Table 2 summarises the compression ratios achieved by BZIP, GZIP, and compress. It enables us to evaluate our compression efforts.

### 3.2 The Elements of T-Prefix Matrices

The process of T-decomposition creates a T-prefix matrix and a list of T-expansion parameters. Both need to be saved in order to be able to restore the original file. Detailed knowledge of their internal structures is essential if the compressed file shall be as small as possible. It concerns in particular what kind of values the entries are and how they are distributed within the matrix. In this section, we will analyse T-prefix matrices and T-expansion parameters. For a better understanding, we will illustrate the theory with the help of statistical data calculated from the files of the Calgary Corpus. As we remember, T-prefix matrices are square matrices with the following

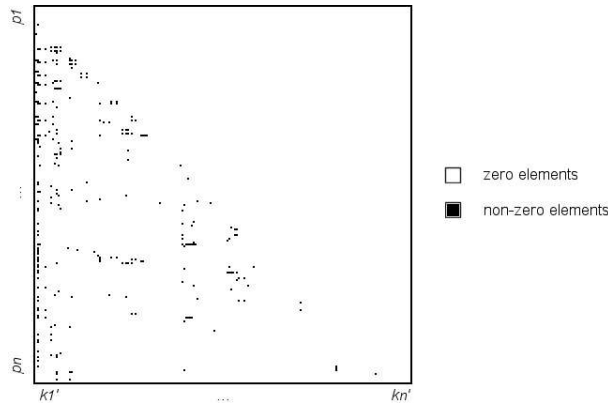
layout:

$$\begin{array}{c|ccccc}
 - & k'_0 & k'_1 & \cdots & k'_{n-2} & k'_{n-1} \\
 \hline
 p_1 & k'_{1,0} & k'_{1,1} & \cdots & k'_{1,n-2} & k'_{1,n-1} \\
 p_2 & k'_{2,0} & k'_{2,1} & \cdots & k'_{2,n-2} & k'_{2,n-1} \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
 p_n & k'_{n,0} & k'_{n,1} & \cdots & k'_{n,n-2} & k'_{n,n-1}
 \end{array} \tag{18}$$

The column  $k'_0$  may contain numerous different symbols while the remaining columns are filled with numbers only. Thus, we call the column  $k'_0$  the *literal vector*. The literal vector merely consists of the source file's literals. We have decided to save the literal vector separately from the remaining matrix entries because of its differing kind of values. The knowledge about the construction of T-prefix matrices gives us other important hints: Only non-negative integers may occur since each row  $p_i$  describes a linear combination of the rows  $p_0$  to  $p_{i-1}$  plus a literal character. Furthermore, all elements above the main diagonal are zero. Accordingly, row  $p_1$  merely contains the literal character. Saving the literal separately, the row would only contain zeros. We may cross out the whole row. That leads to a new, less general layout of T-prefix matrices:

$$\begin{array}{c|ccccc}
 - & k'_1 & k'_2 & \cdots & k'_{n-2} & k'_{n-1} \\
 \hline
 p_2 & k'_{2,1} & 0 & \cdots & 0 & 0 \\
 p_3 & k'_{3,1} & k'_{3,2} & \cdots & 0 & 0 \\
 \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
 p_{n-1} & k'_{n-1,1} & k'_{n-1,2} & \cdots & k'_{n-1,n-2} & 0 \\
 p_n & k'_{n,1} & k'_{n,2} & \cdots & k'_{n,n-2} & k'_{n,n-1}
 \end{array} \quad \text{with } k'_{i,j} \in \mathbb{N}_0 \tag{19}$$

A T-prefix  $p_i$  usually contains only very few of the previous prefixes  $p_0$  to  $p_{i-1}$ . In other words, numerous digits of any row  $p_i$  are zero. The result, a matrix with many zero-elements, is called a *sparse matrix*. The few non-zero elements are uneven distributed:



file name	file size (bytes)	# $p_i$	# $C_{occ}$	# $k_i$ > 1	# S	# $k'_{i,j}$		
						= 1	= 2	> 2
bib	111261	10663	2710	5	81	42618	155	0
book1	768771	78071	11820	15	82	331764	592	2
book2	610856	56766	11330	20	96	235367	1058	25
geo	102400	19570	2375	5	256	44262	462	0
news	377109	39548	9635	30	98	146338	637	629
obj1	21504	3869	866	7	256	8004	521	157
obj2	246814	25629	7733	38	256	84752	461	166
paper1	53161	6685	1769	5	95	24385	41	2
paper2	82199	9969	2103	6	91	36768	12	2
paper3	46526	6461	1432	1	84	21677	2	0
paper4	13286	2186	604	2	80	6620	6	0
paper5	11954	2051	588	3	91	6008	3	0
paper6	38105	5054	1469	6	93	17771	24	5
pic	513216	15585	2697	24	159	35929	1255	5307
progc	39611	5185	1560	9	92	17832	87	1
progl	71646	5996	2119	7	87	24436	369	16
progp	49379	4338	1846	4	89	16187	0	0
trans	93695	6387	2969	14	99	26986	226	126

Table 3: Calgary Corpus' files. File sizes in bytes. # $p_i$ , number of T-prefixes obtained through T-decomposition of the benchmark files. # $C_{occ}$ , number of occupied columns in the corresponding T-prefix matrix. # $k_i > 1$ , number of T-expansion indices greater than one. # S, cardinality of the underlying alphabet S. # $k'_{i,j} = 1$ , number of ones in the matrix. # $k'_{i,j} = 2$ , number of twos in the matrix. # $k'_{i,j} > 2$ , number of entries greater than two.

### Definition 3.1 (Occupied Column)

A column  $k'_j$  with  $j \in \{1, \dots, n-1\}$  is called **occupied** if it contains at least one non-zero element. Otherwise, we call that column unoccupied or empty.

Table 3 offers an insight into the T-prefix matrices' values. Although the files of the Calgary Corpus differ significantly, there are similarities in the structure of the appropriate T-prefix matrices. A T-prefix matrix has approximately  $\frac{(\#p_i)^2}{2}$  elements. All of the T-prefix matrices are very sparse, which may be seen in the numbers of non-zero elements compared to the total numbers of elements. Furthermore, most of the non-zero elements are one. Only few of them are two or even greater.

Additionally to the T-prefix matrices, the T-expansion parameters need to

be stored. The T-expansion parameters  $k_i$  indicate how often the according prefix is repeated in the source file. Most T-expansion parameters are of small value because the input files usually follow patterns which seldom repeat themselves more than a few times in a row. That is, pictures rarely contain large areas of the same colour, texts rarely contain words or sentences that repeat themselves several times in a row. In fact, very few of the T-expansion parameters are greater than one (see table 3).

Finally, the literal vector is to be saved. It may consist of all possible characters that may be found in a file. For numerous files merely use a small range of symbols, the cardinality of the used alphabet  $S$  tends to be small.

## 4 Compression Approaches

### 4.1 T-Prefix Matrix

#### 4.1.1 Store the Whole Matrix

Our first approach was to store T-prefix matrices completely. The new files reached enormous sizes. We did not expect them to be that huge. The version of the compression utility was useless for the purpose of data compression, as it saved *every* element of the matrix. Consider the file `paper1` of the Calgary Corpus, it has 6,685 T-prefixes. Hence, the T-prefix matrix has nearly 22,500,000 elements. Even if we had been able to store each element in one bit, the output would have had a size of about  $2.7MB$ . Rather bad, compared to the original file with a size of  $53kB$ . Accordingly, we did not benchmark this version.

However, we used that program to obtain statistical data on T-prefix matrices, literal vectors, and T-expansion indices.

#### 4.1.2 Store the Matrix Column by Column

A T-prefix matrix is usually very sparse and the non-zero elements are uneven distributed, i.e., many columns are not occupied. Removing these empty columns before encoding a T-prefix matrix reduces the amount of data to be stored. We encoded the T-prefix matrix column by column beginning with  $k'_1$ . Each occupied column got a small header that contained the number of columns skipped before that one and the number of bits used to encode zero-runs between the column's values themselves. Each of a column's ones or twos was described by a zero-run, i.e., its position, and one bit for its value. Larger values are too rare to always reserve enough bits for them. We stored the entries  $k'_{i,j} > 2$  as triples  $(i, j, k'_{i,j})$ .

<i>Element</i>	<i>Bits reserved</i>	<i>Description</i>
Header		
<i>ID</i>	3	Version-ID of encoding scheme.
<i>Width</i>	32	Width of the T-prefix matrix, the number of rows.
T-expansion parameters		
<i>TEP<sub>Num</sub></i>	10	Number of T-expansion parameters greater than one.
<i>TEP<sub>Bits</sub></i>	5	Number of bits used to save the positions of the parameters.
<i>repeat TEP<sub>Num</sub> times</i>		
<i>i</i>	<i>TEP<sub>Bits</sub></i>	Position of that parameter.
<i>FLAG<sub>1</sub></i>	1	If zero, the value is two and the next position $k_i$ follows. If one, go to <i>FLAG<sub>2</sub></i> .
<i>FLAG<sub>2</sub></i>	1	If zero, read 5-bit value ( $l = 5$ ). If one, read 32-bit value ( $l = 32$ ).
$k_i$	$l$	Value of the current expansion parameter.
Literals		
$k'_{i,0}$	8 bit	ASCII-coded literals, <i>Width</i> times.
Triples		
<i>TRI<sub>Num</sub></i>	32	Number of triples saved.
<i>TRI<sub>Bits</sub></i>	-	Bits used to encode the positions: = LOG2( <i>Width</i> ).
<i>repeat TRI<sub>Num</sub> times</i>		
<i>i</i>	<i>TRI<sub>Bits</sub></i>	The triple's position
<i>j</i>	<i>TRI<sub>Bits</sub></i>	in the matrix.
<i>FLAG<sub>1</sub></i>	1	If zero, read 4-bit value ( $l = 4$ ). If one, read 32-bit value ( $l = 32$ ).
$k'_{i,j}$	$l$	Value.
T-prefix matrix		
<i>CM<sub>Bits</sub></i>	5	Number of bits used to encode the distance between to occupied columns.
<i>OCC</i>	30	Number of occupied columns.
Columnheader		
<i>skipped</i>	1	If zero, this column is adjacent to the previous one. Go to <i>FLAG<sub>2</sub></i> .
<i>FLAG<sub>1</sub></i>	1	If zero, $l = \text{Log}_2(\text{Width}/100)$ . If one, distance to previous column is long. set $l = \text{CM}_{Bits}$ .
<i>FLAG<sub>2</sub></i>	1	If one, read <i>BITLEN</i> .
<i>BITLEN</i>	4	Number of bits used to encode distances between non-zero elements.
Columnbody		
<i>DST</i>	<i>BITLEN</i>	Distance to previous element in this column. $DST = 0$ means end of column.
$k'_{i,j}$	1	If zero, the value is one. If one, the value is two.

Table 4: Detailed description of the file format used to store T-prefix matrices column by column.

<i>file name</i>	<i>Format as described in table 4</i>	<i>Same format, but Huffman-encoded literals</i>
bib	0.623	0.605
book1	0.726	0.686
book2	0.711	0.681
geo	0.860	0.859
news	0.742	0.715
obj1	0.838	0.850
obj2	0.750	0.744
paper1	0.776	0.740
paper2	0.725	0.685
paper3	0.764	0.720
paper4	0.794	0.750
paper5	0.834	0.799
paper6	0.802	0.769
pic	0.215	0.206
progc	0.785	0.753
progl	0.590	0.564
progp	0.591	0.569
trans	0.549	0.533

Table 5: Results of the column-based approach. All values bits per bit.

Furthermore, we saved only those T-expansion parameters which are greater than one. Each double consisted of the parameter’s position and value  $(i, k_i)$ . The technical details of that encoding scheme may be found in table 4. Another point of interest were the literals. The version stored them using 8 bits per literal, without regarding their number and distribution. Using a fixed-length scheme, only  $\log_2(n)$  bits were needed to encode a literal uniquely, where  $n$  is the number of different literals. A variable-length scheme additionally considered the frequency of each literal. Particularly on (text) files with smaller alphabets, it could be a way to avoid wasting storage capacity. Accordingly, we used static Huffman encoding for the literals in another version. The T-prefix matrix and T-expansion parameters were saved as before.

## Conclusion

Table 5 shows the results of both versions. All benchmark files could be

converted into smaller files. Accordingly, the approach reached the aim to compress the given files. The second version achieved slightly better results than the first one. Thus, we decided to prefer the Huffman-encoding scheme for the literals in other approaches as well.

The main problem of this technique were the headers used for each column, however. They were only meta information consuming space without containing any information about the matrix itself. Therefore a new objective was to find a scheme that needed less meta information. Even better would be a canonical scheme, i.e., required meta information could be obtained from an already processed part of the matrix and need not be separately saved.

### 4.1.3 Store the Matrix As A Single Vector

To consider a given T-prefix matrix as a single vector was close to the previous approach. We had to deal with a lower triangle matrix as seen in formula 19. We obtained one only vector through appending each column to the previous one. The process may be illustrated by the consecutive numbering of a given matrix' elements:

$$\begin{array}{c|cccccc}
 - & k'_1 & k'_2 & \cdots & k'_{n-2} & k'_{n-1} \\
 \hline
 p_2 & 1 & & & & \\
 p_3 & 2 & n & & & \\
 \vdots & \vdots & \vdots & \ddots & & \\
 p_{n-1} & n-2 & 2n-4 & \cdots & \frac{n(n-1)}{2} - 2 & \\
 p_n & n-1 & 2n-3 & \cdots & \frac{n(n-1)}{2} - 1 & \frac{n(n-1)}{2}
 \end{array} \tag{20}$$

$$\begin{pmatrix} k'_1 \\ \\ \\ k'_2 \\ \\ \vdots \\ k'_{n-2} \\ k'_{n-1} \end{pmatrix} \iff \begin{pmatrix} 1 \\ 2 \\ \vdots \\ n-1 \\ n \\ \vdots \\ 2n-3 \\ \vdots \\ \frac{n(n-1)}{2} - 2 \\ \frac{n(n-1)}{2} - 1 \\ \frac{n(n-1)}{2} \end{pmatrix} \tag{21}$$

We mentioned that we could compute the length of each row or column with the help of knowing the dimension of the matrix. Regarding the enumeration scheme above, it was possible to reobtain the original matrix from the

created vector. The conversion was reversible.

The approach had the advantage that we needed to store only one number instead of two in order to identify a value's position in the matrix. Unfortunately, it resulted in another problem: The index for a value in the vector could be a number up to  $\frac{n(n-1)}{2}$ . In contrast, the maximum size for an index of a column or row in the matrix is  $n - 1$ . We again used the description of zero-runs. The average distance to be stored was about  $\frac{n(n-1)}{2^{*(\#k'_{i,j} > 0)}}$  where  $n$  was the number of T-prefixes and  $\#k'_{i,j} > 0$  was the number of values in the matrix that were greater than zero. Nevertheless, the maximum distance remained at a size of around  $\frac{n(n-1)}{2} - (\#k'_{i,j} > 0)$  because of the uneven distribution of the matrix' elements. In order to decrease the maximum distance, we only saved occupied columns and skipped the empty ones. Hence, T-prefix matrices were processed in two steps. First, we created a binary index vector `idx` and stored for each column  $k'_j$  whether it was occupied (`idx[j]=1`) or not (`idx[j]=0`). In the second step, we concatenated the occupied columns. The result was a vector denser and smaller than the original T-prefix matrix. Due to that improvement, the maximum possible distance was declined to  $2n - 2$ , which would be the distance from the very first value to a value at the end of the second column. That distance decreased from column to column and could easily be computed with the help of the index vector `idx`. Accordingly, we significantly dropped the storage capacity we needed to reserve for each value's position. Nevertheless, we would have wasted a lot of memory capacity, if we had used the same number of bits for both small and large numbers. The maximum distance, for instance, was 1000 which needed 10 bits for saving. In contrast, the current distance could be 25 which we would rather store in 5 bits. Therefore, we set some bits before saving the actual value. Details may be seen in table 6.

By now, we knew the position of each of the matrix' elements greater than zero. The last thing to do was to identify all entries with values greater than one and assign their original values. We used the same concept again: We numbered the elements greater than zero and additionally marked all elements greater than one. We then saved both the marked elements' values and the distances between them.

The rising amount of meta information however remained a problem. Therefore, we tried to figure out what potential the approach had and created an optimal version. We saved all occurring distances in the appropriate number of bits. It led us to the minimal possible size of the compressed file. Actually, it is even below the minimum as we were not able to reobtain the original file from the compressed file. We would have to add meta information or precisely define the layout of the file, such that we would not reach the level

<i>Element</i>	<i>Bits used to store</i>	<i>Description</i>
Header		
<i>ID</i>	3	Version-ID of encoding scheme.
<i>Width</i>	32	Width of the T-prefix matrix, the number of rows.
T-expansion parameters		
<i>TEP<sub>Num</sub></i>	10	Number of T-expansion parameters greater than one.
<i>TEP<sub>Bits</sub></i>	5	Number of bits used to save the positions of the parameters.
<i>repeat TEP<sub>Num</sub> times</i>		
<i>i</i>	<i>TEP<sub>Bits</sub></i>	Position of that parameter.
<i>FLAG<sub>1</sub></i>	1	If zero, the value is two and the next position <i>i</i> follows. If one, go to <i>FLAG<sub>2</sub></i> .
<i>FLAG<sub>2</sub></i>	1	If zero, read 5-bit value ( $l = 5$ ). If one, read 32-bit value ( $l = 32$ ).
<i>k<sub>i</sub></i>	<i>l</i>	Value of the current expansion parameter.
Literals		
The literals are encoded with a static Huffman code.		
T-prefix matrix		
<i>LEN<sub>Bits</sub></i>	5	Number of bits used to store the number of matrix entries greater than zero.
<i>LEN</i>	<i>LEN<sub>Bits</sub></i>	Number of matrix entries greater than zero.
<i>Idx</i>	<i>Width</i>	Index vector for occupied columns. $bit_i = 1$ : column <i>i</i> occupied.
<i>MAX<sub>Bits</sub></i>	–	Maximum number of bits needed to store the largest possible distance (automatically computed with the help of the index vector).
<i>repeat LENGTH times</i>		
<i>FLAG<sub>1</sub></i>	1	If zero, go to <i>Flag<sub>2</sub></i> . If one, distance to previous entry is one.
<i>FLAG<sub>2</sub></i>	1	If zero, go to <i>Flag<sub>3</sub></i> . If one, read <i>l</i> -bit value ( $l = 3$ ).
<i>FLAG<sub>3</sub></i>	1	If zero, read <i>l</i> -bit value ( $l = 6$ ). If one, read <i>l</i> -bit value ( $l = MAX_{Bits}$ ).
<i>DST</i>	<i>l</i>	Distance between two matrix elements ( $k'_{i,j} > 0$ ).
<i>assignment of values greater than one</i>		
<i>NUM<sub>Bits</sub></i>	5	Number of bits used to store the number of matrix entries greater than one.
<i>NUM<sub>Elem</sub></i>	<i>NUM<sub>Bits</sub></i>	Number of matrix entries greater than one.
<i>VAL<sub>Bits</sub></i>	5	Number of bits needed for the maximum value.
<i>repeat NUM<sub>Elem</sub> times</i>		
<i>FLAG<sub>1</sub></i>	1	If zero, read <i>l</i> -bit value ( $l = 3$ ). If one, read <i>l</i> -bit value ( $l = LOG_2(LEN - 8)$ ).
<i>DST</i>	<i>l</i>	Distance between two elements greater than one.
<i>FLAG<sub>2</sub></i>	1	If zero, value is two. If one, go to <i>Flag<sub>3</sub></i> .
<i>FLAG<sub>3</sub></i>	<i>VAL<sub>Bits</sub></i>	The element's value.

Table 6: Detailed description of the file format used to store T-prefix matrices as single vectors.

<i>file name</i>	<i>normal encoding</i>	<i>maximal encoding</i>	<i>file name</i>	<i>normal encoding</i>	<i>maximal encoding</i>
bib	0.595	0.364	paper3	0.665	0.420
book1	0.694	0.397	paper4	0.659	0.438
book2	0.623	0.370	paper5	0.671	0.454
geo	0.826	0.541	paper6	0.629	0.407
news	0.668	0.403	pic	-	0.064
obj1	0.721	0.522	progc	0.631	0.409
obj2	0.577	0.371	progl	0.453	0.287
paper1	0.636	0.403	progp	0.453	0.293
paper2	0.640	0.396	trans	0.425	0.272

Table 7: Results of the vector-based approach. All values bits per bit.

of the shown minimum.

## Conclusion

Table 7 shows the compression results of both versions. The column "normal encoding" contains the results achieved by the way of compression which enables us to reobtain the original file. The other column shows the results of the "optimal" version.

The compression approach also achieved the aim to significantly compress the original file. The benchmark shows that the reached level of compression is comparable to the one of approach 4.1.2. Considering the minimum possible file size, the approach seemed to be more promising than the previous one. Nevertheless, the results differed slightly, which made it necessary to find completely different approaches.

## 4.2 Arithmetic Coding

Arithmetic coding is a method of statistical encoding, which encodes symbols depending on their probability. The usage of arithmetic coders is the last step of processing a file. It should be used after as much as possible redundancy had been removed from the input data. Details about the internals of arithmetic coding may be found in [2].

### Definition 4.1 (Context)

Let  $S$  be a set of source symbols and  $T = \{T_1, \dots, T_{\#S}\}$  a set of integers. If  $S$  is mapped uniquely to  $T$ ,  $T$  is called a **context**.

<i>Element</i>	<i>Bits used</i>	<i>Description</i>
<i>to store</i>		
Header		
<i>ID</i>	3	Version-ID
<i>Width</i>	32	Width of T-prefix matrix
T-expansion parameters		
$TEP_{Num}$	10	Number of T-Expansion parameters greater than one.
$TEP_{Bits}$	5	Number of bits used to save the positions of the parameters.
		for each T-expansion parameter
$i$	$TEP_{Bits}$	Position of the parameter.
$k_i$	5	Value of the parameter.
T-prefix matrix - arithmetically encoded		

Table 8: General layout of T-compressed files using arithmetic encoding.

Let  $M$  be an arbitrary T-prefix matrix. We may start to encode it using a canonical mapping to the context with the symbols  $(0,1,\dots,N)$ . That context has to handle many small values, very long runs of the same symbol, and must be able to deal with big numbers, too. The bigger a context is, i.e., the more different symbols it contains, the less the resulting compression will be. It seems to be better to create special contexts for each kind of data. Or to have at least one context for very common data elements like 0 and 1 and one context for very unlikely values, e.g., entries  $> 10$ .

#### **Definition 4.2 (Model)**

*The set of all contexts used to encode a block of data is called a **model**.*

Arithmetic coding allows to use different contexts within one coding block. A model often contains several contexts. The decoder must then be able to shift to the proper context using only the information it has already gained from the treated file. The decoding process will lead to incorrect results if the decoder uses a wrong context. In other words, the decoder has to know everything the encoder knows, at any state. Therefore, it is necessary either to encode meta information containing the used context. Or to shift between contexts using canonical rules which base on the already processed data.

The compression ratio depends on the coding model. A convenient trade-off between many very small and a few big models has to be found. Using only a few models reduces the amount of meta information, i.e., which context is to be used for correct decoding. But the contexts then have to be more general

<i>file name</i>	<i>Initial model</i>			<i>Changed model</i>	
	<i>Threshold</i>			<i>Threshold</i>	
	<i>1</i>	<i>2</i>	<i>8</i>	<i>1</i>	<i>2</i>
bib	0.573	0.581	0.622	0.578	0.586
book1	0.630	0.639	0.686	0.638	0.646
book2	0.583	0.591	0.632	0.589	0.597
geo	0.753	0.761	0.799	0.768	0.775
news	0.621	0.628	0.672	0.628	0.636
obj1	0.715	0.723	0.764	0.726	0.735
obj2	0.555	0.562	0.592	0.564	0.572
paper1	0.630	0.641	0.685	0.639	0.649
paper2	0.629	0.638	0.687	0.636	0.646
paper3	0.660	0.671	0.721	0.669	0.680
paper4	0.668	0.681	0.727	0.677	0.690
paper5	0.680	0.693	0.738	0.691	0.704
paper6	0.628	0.640	0.681	0.637	0.649
pic	0.132	0.134	0.142	0.135	0.136
progc	0.632	0.643	0.685	0.640	0.651
progl	0.460	0.470	0.499	0.467	0.476
progp	0.460	0.468	0.499	0.466	0.474
trans	0.423	0.430	0.456	0.428	0.435

Table 9: Arithmetic coding. A small and simple coding model with thresholds 1, 2, and 8. A small and simple coding model skipping empty columns with thresholds 1 and 2. All values bits per bit.

and may not contain enough information of the matrix' characteristics. The following models handle the matrix in the same way. They differ only in the used contexts. The matrix is regarded as a single vector, see section 4.1.3 for details. The general file format is explained in table 8.

### A Small and Simple Coding Model

Our first approach of arithmetic encoding used a simple and small coding model as suggested by Peter Fenwick[3]. It encoded short runs of length  $n$  of an element  $i$  directly by repeating the symbol  $i$   $n$  times. A separate coding model which encoded the length as a binary number was used for longer runs.

T-prefix matrix - arithmetically encoded		
context	element	meaning
<i>primary</i>	1	Encodes $k'_{i,j} = 1$ .
	2	Switches to context <i>number</i> to encode a $k'_{i,j} > 1$ .
	3	Switches to context <i>number</i> to encode a run of zeroes longer than <i>Threshold</i> .
<i>number</i>	0	Encodes a zero-bit of the run length.
	1	Encodes a one-bit.
	2	Encodes the most significant bit and switches back to context <i>primary</i> .
<i>literals</i>	0..255	numerical value of each literal

A static threshold sets the maximal run length for direct encoding. The threshold hence affects the compression as well, see table 9.

Skipping empty columns and indicating them in a “dictionary” was very similar to the column-based approaches discussed in sections 4.1.2 and 4.1.3. Using the above model, many runs would be much shorter without the empty columns. Despite having skipped unoccupied columns the compression did not improve as we may see in table 9.

Larger models may take more care of the structure of a T-prefix matrix, but we would have to save more meta-information to encode the used context. Therefore a trade-off between too few and too many contexts must be found. Models could take into account that short runs frequently appear, longer runs regularly, and very long runs rarely. A context that encodes very frequent short runs must be small, while one that encodes rare long runs may be bigger.

## A Coding Model of Intermediate Size

In the previous scheme, every run was encoded as a sequence of numbers. That representation for long runs was very useful because it was universal and did not limit the range of numbers. Nevertheless, there were more short runs than long runs to encode since the distribution of non-zero elements in T-prefix matrices was usually uneven. Consequently, the encoding scheme should regard that and handle the frequent short runs more compact. The following encoding scheme directly encoded runs of length 0 to 9, which were the most common run lengths. Table 10 shows the results of that optimized approach. The compression ratio has increased compared to the small model. Thus it is worth to handle different run lengths with encoding schemes which consider their frequency and structure.

<i>file name</i>	<i>medium</i>	<i>big</i>	<i>file name</i>	<i>medium</i>	<i>big</i>
bib	0.500	0.503	paper3	0.571	0.575
book1	0.550	0.557	paper4	0.589	0.593
book2	0.510	0.516	paper5	0.604	0.607
geo	0.691	0.688	paper6	0.558	0.562
news	0.545	0.551	pic	0.120	0.122
obj1	0.635	0.638	progc	0.557	0.562
obj2	0.495	0.499	progl	0.340	0.403
paper1	0.555	0.558	progp	0.401	0.405
paper2	0.543	0.547	trans	0.376	0.378

Table 10: Arithmetic coding. Results of a coding model of intermediate size and of a big model. All values bits per bit.

T-prefix matrix - arithmetically encoded		
context	element	meaning
<i>primary</i>	0-9	Encodes a run of that length.
	10	Switches to context <i>number</i> to encode a run longer than 9.
	11	Switches to context <i>number</i> to encode a $k'_{i,j} > 1$ (all $k'_{i,j} = 1$ need not be saved, as $k'_{i,j} = 1$ is implied in each entry having no value).
<i>number</i>	0	Encodes a zero-bit of the run length.
	1	Encodes a one-bit.
	2	Encodes the most significant bit and switches back to context <i>primary</i> .
<i>literals</i>	0..255	numerical value of each literal

## A Big Coding Model

As seen in the previous model, grouping the runlengths and encoding them differently may result in better compression. The next model used three contexts: one for short runs, one to encode runs of intermediate length and binary encoding for long runs. It also pointed out the main problem: What were appropriate limits between the domains? We defined static limits. The encoding scheme is presented below.

T-prefix matrix - arithmetically encoded		
context	element	meaning
<i>primary</i>	0	Switches to context <i>number</i> to encode a $k'_{i,j} > 1$ .
	1	Switches to context <i>short_run</i> to encode a run shorter than 11.
	2	Switches to context <i>medium_run</i> to encode a run longer than 10 and shorter than 201.
	3	Switches to context <i>number</i> to encode a run longer than 200.
<i>number</i>	0	Encodes a zero-bit of the run length.
	1	Encodes a one-bit.
	2	Encodes the most significant bit and switches back to context <i>primary</i> .
<i>short_run</i>	0-10	Encodes runs of length 0..10.
<i>medium_run</i>	0-190	Encodes runs of length 11..200.

By setting the domains dynamically depending on the T-prefix matrix, the model would regard the specific characteristics of each source file much better. But we then had to store these parameters, too. That optimization problem has not been solved yet. The achieved results are shown in table 10.

Compared to the intermediate model, this encoding scheme achieves slightly less compression on all files but *geo*. That could be caused either by the overhead which resulted from the meta information needed to shift to the convenient context. Or by an unfortunate choice of the domains for short, intermediate and long runs. Regarding the marginal difference of the current and previous results, this coding model could have some potential.

## Conclusion on Arithmetic Coding

All these models may be improved for two purposes: First, they were static. That means they did not adapt themselves to the matrix which was to be compressed. Particularly the last model contained some parameters that could be set dynamically, e.g., the size of both the small and the medium context. Another weakness was the way of saving the literals. A set of 256 different symbols was supported in our approaches, which was rarely required.

<i>file name</i>	<i>coverage %</i>	<i>file name</i>	<i>coverage %</i>
bib	0.3139	paper3	0.3003
book1	0.1484	paper4	0.4340
book2	0.2586	paper5	0.4654
geo	0.5576	paper6	0.4506
news	0.2701	pic	0.3724
obj1	0.5639	progc	0.4129
obj2	0.4869	progl	0.3301
paper1	0.3512	progp	0.4199
paper2	0.2494	trans	0.4227

Table 11: Average coverage of maximal matchings.

### 4.3 Prefix Coding

In this approach we tried to reduce the redundancy of a T-prefix matrix with a deeper knowledge of T-depletion codewords. We investigated whether there were patterns in a T-prefix matrix that repeated. The T-depletion codeword (without the literal symbol) of an arbitrary T-prefix  $p_i$  could frequently be expressed as the combination of the T-depletion codeword (except the literal character) of a prefix  $p_h$  from an above row and some additional elements  $k'_{i,l}$ ,  $l = h, \dots, i - 1$ .

$$\underbrace{(k'_{i,1}, k'_{i,2}, \dots, k'_{i,i-1})}_{\text{row}(p_i)} = \underbrace{(k'_{h,1}, k'_{h,2}, \dots, k'_{h,y})}_{\text{row}(p_h)_y} k'_{i,y+1}, \dots, k'_{i,i-1} \quad (22)$$

with  $1 \leq y < h < i \leq n$ .

#### Definition 4.3 (Matching and Matching Remainder)

Let  $\text{row}(p_h) = (k'_{h,1}, k'_{h,2}, \dots, k'_{h,h-1})$  and  $\text{row}(p_i) = (k'_{i,1}, k'_{i,2}, \dots, k'_{i,i-1})$  with  $2 \leq h < i \leq n$  be the numeric parts of two T-depletion codewords belonging to the T-prefixes  $p_h$  and  $p_i$ .  $\text{row}(p_h)_y = (k'_{h,1}, k'_{h,2}, \dots, k'_{h,y})$  with  $y < h$  is called a **matching** of length  $y$  for  $\text{row}(p_i)$  iff  $k'_{h,x} = k'_{i,x} \forall x \in \{1, \dots, y\}$ . The vector of the remaining elements  $\text{row}(p_i)_{y+} = (k'_{i,y+1}, \dots, k'_{i,i-1})$  is called the **matching remainder**.

#### Definition 4.4 (Maximal Matching)

Let  $\text{row}(p_h)_y$  be a matching for  $\text{row}(p_i)$ .  $\text{row}(p_h)_y$  is called a **maximal matching** for  $\text{row}(p_i)$  iff there is no other matching  $\text{row}(p_l)_z$ ,  $l < i$  with  $z > y$ .

<i>filename</i>	<i>bits per bit</i>	<i>filename</i>	<i>bits per bit</i>
bib	0.497	paper3	0.668
book1	0.599	paper4	0.699
book2	0.558	paper5	0.731
geo	0.963	paper6	0.626
news	0.602	pic	0.151
obj1	0.846	progc	0.633
obj2	0.585	progl	0.423
paper1	0.621	progp	0.429
paper2	0.613	trans	0.354

Table 12: Results of prefix coding.

We then calculated the average length of the maximal matchings for each file of the Calgary Corpus, see table 11. In order to gain compression from the redundance of the T-depletion codewords, we encoded each row of a T-prefix matrix  $row(p_i)$  according to the following scheme:

Length $y$ of the maximal matching $row(p_h)_y$ .	Index $h$ of the maximal matching.	Matching remainder $row(p_i)_{y+}$ , run length encoded (arithmetic coding).
---	--	--

## Conclusion

Table 12 shows the results of the approach. The use of maximal matchings too achieved the objective to significantly compress the original files. The results are similar to those of the previous attempts.

## 4.4 Using T-Prefix Matrices as Preprocessors

The conversion of a file into a T-prefix matrix discovers and removes redundant information. A matrix represents the source file in a strongly changed way. The transformation might be an effective preprocessing to enhance the compression achieved by another compression algorithm.

We thus developed different concepts how to use T-Codes as a preprocessor. The results of each technique is summarized in table 13.

**A Single Vector** The easiest was to regard the whole T-prefix matrix as a single vector, see section 4.1.3. The resulting string then was processed

<i>file name</i>	<i>Single vector</i>		<i>Index vector</i>	
	<i>GZIP</i>	<i>BZIP</i>	<i>GZIP</i>	<i>BZIP</i>
obj1	0.884	0.676	0.853	0.733
paper1	0.881	0.655	0.805	0.636
paper2	0.831	0.635	0.777	0.623
paper3	0.936	0.684	0.831	0.668
paper4	0.879	0.732	0.806	0.707
paper5	0.897	0.751	0.830	0.739
paper6	0.878	0.658	0.801	0.649
progc	0.885	0.673	0.808	0.658
progl	0.624	0.480	0.570	0.465
progp	0.632	0.492	0.568	0.482
trans	0.576	0.433	0.549	0.415

Table 13: Results of different concepts using T-Codes for preprocessing. All values bits per bit.

with another compressor.

It was a theoretical approach only. We wrote the whole T-prefix matrix into a file using the minimal number of bits for each value without any meta information. So the process was not reversible.

**A Vector With A Column Index** The concept of using an index vector to avoid storing empty columns was introduced in section 4.1.3. Like in the previous concept, we encoded all values with the minimal number of bits required, but skipped unoccupied columns.

**A Run Length Encoded Vector** A representation similar to the coding models introduced in section 4.2 could be created and compressed by GZIP or BZIP. The output of the arithmetic coders did not contain enough redundancy to compress it again. The file sizes increased and were not taken into account.

## Conclusion

Using T-depletion codes as intermediate step for common compression tools was also able to compress the original files. Nevertheless, the compression ratio was not satisfying. Especially as we used strings without any meta information which would have to be added in order to establish a reversible algorithm. In other words, there was no need to pay further attention to

that approach.

## 4.5 Contiguous Range Indices

The contiguous range index conversion, see section 2.5, removes redundancy from the T-coding tree. Therefore the total amount of information to store is less than in the previous approaches, which may result in a better compression.

The contiguous range index conversion generates very large numbers, which cannot be handled with the build-in types of C++. Our implementation contained a valid algorithm for converting T-depletion codes into contiguous range indices and vice versa. Unfortunately it was able to process files of a few hundred bytes only because `long int` was limited to  $2^{32}$ . We therefore could not gain any statistical data regarding the possible compression.

Floating point numbers may be used to process and save contiguous integer indices. But that will not solve the problem of dealing with possibly huge numbers in general. The values gained from numerous files are even to large for `float` oder `long double`.

Many contiguous range indices seem to be close to powers of two. That provides the possibility of processing and saving each value  $I(x)$  as  $I(x) = 2^p + q$  where only  $p$  and  $q$  need to be saved. That sounds rather simple, but there are a few exceptions which have to be regarded. Frequently, contiguous range indices are small numbers as the belonging T-depletion codeword contains only  $k'_{i,j} > 0$  with small  $i, j$ . Next, we deal with the structure of contiguous integer indices and then develop a suitable representation scheme for them. The number of leaf nodes  $\#S_i$  of a T-Code tree at T-augmentation level  $i$  may be found with the help of equation 13, which may be written as

$$\#S_i = \#S_{i-1}(k_i + 1) - k_i. \quad (23)$$

We may see that  $\#S_i$  basically depends on the T-expansion parameters  $k_i$ . A closer look to equation 14 reveals that  $I_{m+1}(p_1) = I_m(p_i) + k'_{i,m+1}(\#S_m - 1)$  may be a useful approximation to analyse the structure of contiguous range indices. We changed that to  $I_{m+1}(p_i) = I_m(p_i) + k'_{i,m+1}\#S_m - k'_{i,m+1}$  and inserted equation 23:

$$I_{m+1}(p_i) = I_m(p_i) + k'_{i,m+1}(k_m + 1)\#S_m - k'_{i,m+1}(k_m + 1) \quad (24)$$

Regarding the second addend,  $I_{m+1}(p_i)$  depends very much on the two factors  $k'_{i,m+1}$  and  $k_m + 1$ . Suppose  $k'_{i,j+1} = 2^x$  and  $k_j = 2^x - 1$  with  $j = 1, \dots, m$  and  $x \in \mathbb{N}$ ,  $I_{m+1}$  may be close to a power of 2 and stored in a space-saving way. Otherwise,  $I_{m+1}$  may be close to the power of another base. Then

the base has to be either stored in the file, or calculated from the available parameters, which we did not look for.

Deeper knowledge of the structure is needed to find both a scheme for handling contiguous integer indices during calculations and for saving them conveniently.

## 5 Discussion

Overall, we have shown that it was possible to compress data files with the help of T-Codes. The approaches differed significantly as seen in section 4. And so did the results, which are compared in figure 1.

Approach 4.1.1, which stored the whole matrix, was the worst. Redundant information was not removed, which resulted in the need of extensive storage capacity. Nevertheless, it helped to gain information on T-prefix matrices especially the distribution of their entries.

We then decided to try two column-based approaches (4.1.2, 4.1.3). It turned out that they were rather useful. The two of them achieved good results. Both approaches may also be improved although they will not become competitive to other leading compression algorithms. That may be seen in figure 1: we benchmarked a kind of optimal version, which was not reversible. It was still unable to close the gap to the leading programs.

In contrast, arithmetic coding, see section 4.2, was a quite different approach. It reached better results as the previous two approaches. It also offers opportunities for improvements, which means to find better models than the investigated and shown ones.

Prefix coding, see section 4.3, then connected parts of the previous approaches. But its results were average. The approach could still be improved. On the one hand the run length encoding could be further researched. On the other hand there might be some other possibilities to store length and index of the maximal matchings more conveniently. But it seemed to us that the approach heavily depended on the used arithmetic coder. And the results of arithmetic coding were better. Accordingly, arithmetic coding may always achieve better results than prefix coding if both use the same model.

The next approach achieved less satisfying results. They show that using a T-prefix matrix as a preprocessor did not work very well. It still compressed files but the approach could not even compete with our previous versions though it was a kind of "optimal", i.e., not reversible, version.

The last approach seems to be promising, but it is rather difficult to deal with. Because of the possibly huge size of contiguous range indices, a data structure has to be developed which is able to handle large integers. We

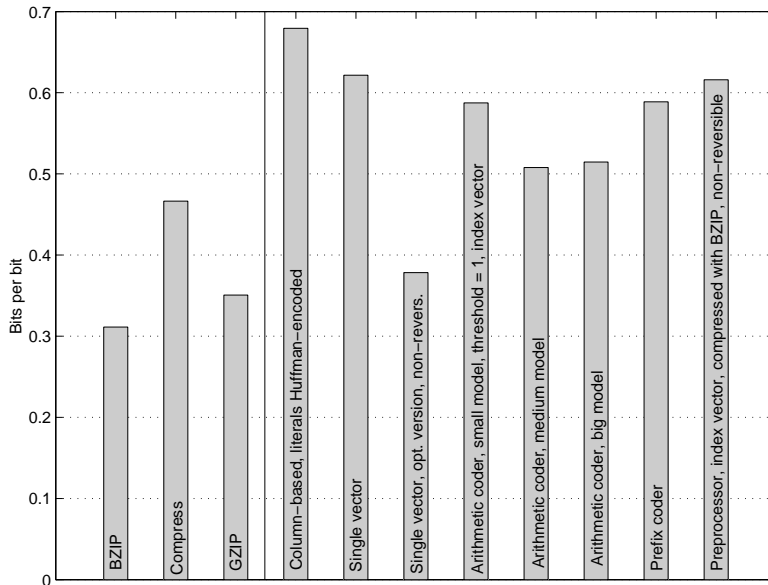


Figure 1: Average benchmark results of both established algorithms and our compression approaches.

may consider a small binary file's T-prefix matrix with about 1000 rows. In addition, we assume all T-expansion parameters to be 1. Thus, we obtain a maximal possible size of about  $2^{1000}$  for the corresponding contiguous integer indices, according to theorem 2.16. Unfortunately, such a data structure is not the whole solution. A suitable way of saving these large numbers is required as well. It seems easy to understand that it is not recommendable to store a number like  $2^{1000}$  in 1000 bits.

All in all, we were able to compress data files with the help of T-Codes. Though they could not bridge the gap to other popular algorithms. Nevertheless, there remain several methods to implement such that the opportunity is not excluded that data compression with T-Codes may become able to compete with leading programs in the field.

## 6 Acknowledgements

We owe special thanks to our supervisor Ulrich Speidel for his support and suggestions, which helped enormously to realise the program. We further would like to thank our families, whose support still made it possible to undertake the project.

## References

- [1] ABEL, J. Calgary Corpus. Website, Oct. 2004. <http://www.data-compression.info/Corpora/CalgaryCorpus/index.htm>.
- [2] ABEL, J. The Data Compression Resource on the Internet. Website, Oct. 2004. <http://www.data-compression.info>.
- [3] FENWICK, P. verbal communication.
- [4] GÜNTHER, U. Data Compression and Serial Communication with Generalized T-Codes. *Journal of Universal Computer Science* 2, 11 (Nov. 1996), 769–795.
- [5] GÜNTHER, U. *Robust Source Coding with Generalised T-Codes*. PhD thesis, The University of Auckland, 1998.
- [6] NICOLESCU, R. Uniqueness Theorems for T-Codes. *Tamaki Report Series*, 9 (Sept. 1995). The University of Auckland.
- [7] SPEIDEL, U. verbal communication.
- [8] TITCHENER, M. Digital Encoding by Means of New T-Codes to Provide Improved Data Synchronization and Message Integrity. *IEE Proceedings Pt. E* 131, 4 (July 1984), 51–53.
- [9] TITCHENER, M. Construction and Properties of Augmented and Binary-Depletion Codes. *IEE Proceedings Pt. E* 132, 3 (May 1985), 163–169.
- [10] TITCHENER, M., AND HUNTER, J. Synchronisation Process for the Variable-Length T-Codes. *IEE Proceedings Pt. E* 133, 1 (Jan. 1986), 54–64.
- [11] YANG, J., AND SPEIDEL, U. An Improved T-Decomposition Algorithm. *Fourth ICICS and IEEE PCM (Singapore)* (Dec. 2003).