Universal Data Compression Based on the Burrows and Wheeler-Transformation: Theory and Practice

Bernhard Balkenhol*
Fakultät für Mathematik
Universität Bielefeld
Postfach 100 131
33501 Bielefeld
Germany

Stefan Kurtz†
Technische Fakultät
Universität Bielefeld
Postfach 100 131
33501 Bielefeld
Germany

September 8, 1998

Abstract

A very interesting recent development in data compression is the Burrows and Wheeler-Transformation [3]. The idea is to permute the input sequence in such a way that characters with a similar context are grouped together. We provide a thorough analysis of the Burrows and Wheeler-Transformation from an information theoretic point of view. Based on this analysis, we systematically consider techniques to efficiently implement a data compression program based on the transformation. We show that our program achieves a better compression rate than other programs that have similar requirements in space and time.

*E-mail: bernhard@mathematik.uni-bielefeld.de

†E-mail: kurtz@techfak.uni-bielefeld.de

1 Introduction and Overview

A very interesting recent development in data compression is the Burrows and Wheeler-Transformation [3]. The idea is to permute the input sequence in such a way that characters with a similar context are grouped together. This property allows a locally adaptive statistical compression scheme to achieve compression rates that are close to the best known rates. However, the important point is that these rates can be achieved with much less computational effort than previous programs based on statistical modeling techniques. Thus data compression based on the Burrows and Wheeler-Transformation is fast and it leads to good compression results.

Up until now the Burrows and Wheeler-Transformation has not been thoroughly analyzed from an information theoretic point of view. One of the main contributions of this paper is to provide such an analysis. Assuming that our information source is modeled by a context tree [27], we will show that the Burrows and Wheeler-Transformation permutes the source in such a way, that the permutation can be partitioned into intervals, one for each leaf of the context tree. Due to the property of the context tree, the subsequence of the source in each such interval is i.i.d. As a consequence, a data compression scheme based on the Burrows and Wheeler-Transformation can (in principle) achieve the same compression rates as any other context tree based method, but with much less space requirement.

Based on these theoretical insights, we systematically consider practical and engineering aspects. That is, we describe techniques to efficiently implement a data compression program based on the Burrows and Wheeler-Transformation. We consider some new techniques as well as some well-known techniques. We always carefully motivate their applicability, possibly modify them, and show how to make them work well in practice. This always includes the analysis of space and time requirements. Our contributions are as follows:

- We discuss in which cases run length encoding should be applied, and in which cases better not.
- We explain why the alphabet should always be encoded, and provide a new efficient alphabet encoding technique.
- We describe how to efficiently construct the Burrows and Wheeler-Transformation in $O(kn)$ worst case time and $12n$ bytes of space, where k is the size of the input alphabet and n is the length of the input sequence. These complexity results are achieved using suffix trees [26] and the implementation techniques of [14].
- We provide a technique to efficiently encode runs of zeros after the move-to-front transformation.

- We describe an hierarchical model to estimate the probability distributions required for arithmetic encoding. Similar to other programs, probabilities are estimated on two levels, based on some statistics. For our application we do not only increment statistics but halve them at different speeds. In this way, we can gradually change contexts. The estimators we obtain are a generalization of the k-array β-biased Dirichlet estimators [22, 23].

We have developed a data compression program that employs these implementation techniques. Experimental results show that it achieves a better compression rate than previous programs for 11 of 14 files of the Calgary Corpus. Our program also has the best average compression rate, namely 2.32 bits/byte. For the same 14 files of the Calgary Corpus our program achieves about 50% of *gzip*'s speed for compression. The space requirement of our program is about $10n$ bytes on average. Further experiments show that our program sometimes clearly outperforms *gzip* in compression rate as well as compression time.

This paper is organized as follows: In Section 2, we take some time to carefully establish some basic notions and standard techniques. In particular, in Section 2.3 we review the idea of arithmetic coding, and show why it should be preferred to Huffman coding for our application. Section 2.4 shows how to estimate unknown parameters using the Krichevsky-Trofimov estimator, and in Section 2.5 we review some basic properties of context trees. Section 3 is devoted to the Burrows and Wheeler-Transformation. We carefully define the transformation and state its properties in Section 3.1. Section 3.2 and 3.3 show how to compute and reverse the transformation in linear worst case time and space. In Section 4, we consider implementation techniques. Finally, in Section 5 we present some experimental results.

2 Preliminaries

2.1 Sequences and Trees

For any $l, r \in \mathbb{N}_0$, $[l, r]$ denotes the set $\{i : l \leq i \leq r\}$. Let ε denote the *empty sequence*. For any set $\mathcal{S}$, we define $\mathcal{S}^0 = \{\varepsilon\}$, and $\mathcal{S}^{i+1} = \{as : a \in \mathcal{S}, s \in \mathcal{S}^i\}$. $\mathcal{S}^* = \bigcup_{i \geq 0} \mathcal{S}^i$ is the *set of sequences over* $\mathcal{S}$. $\mathcal{S}^+$ denotes $\mathcal{S}^* \setminus \{\varepsilon\}$. The *length* of a sequence s, denoted by $|s|$, is the number of elements in s. If $s = uvw$ for some (possibly empty) sequences u, v and w, then u is a *prefix* of s, v is a *factor* of s, and w is a *suffix* of s. A prefix or suffix of s is *proper* if it is different from s.

s_i is the ith element in the sequence s. That is, if $|s| = n$, then $s = s_1 \ldots s_n$ where $s_i \in \mathcal{S}$. $s_n \ldots s_1$, denoted by s^{-1}, is the *reverse* of $s = s_1 \ldots s_n$. If $i \leq j$, then $s_i \ldots s_j$ is the factor of s beginning with the ith element and ending with the jth element. If $i > j$, then $s_i \ldots s_j$ is the empty sequence. A factor v of s

begins at position i and ends at position j in s if $s_i \dots s_j = v$. To conveniently refer to the factors of a sequence, we use the abbreviation s_i^j for $s_i \dots s_j$.

Throughout this paper we assume that $\mathcal{X}$ is a finite ordered set of size k, the *alphabet*. The elements of $\mathcal{X}$ are *characters* or *symbols*. If convenient we denote the characters by their *rank* w.r.t. the order on $\mathcal{X}$, i.e. we write the k characters in $\mathcal{X}$ as $1, \dots, k$. If not stated otherwise, x is a sequence of length n over alphabet $\mathcal{X}$. For any alphabet $\mathcal{X}$, any $x \in \mathcal{X}^*$, and any $a \in \mathcal{X}$, $occ_x(a)$ denotes the number of occurrences of a in x. We define $occ_x(S) = \sum_{a \in S} occ_x(a)$ for any $S \subseteq \mathcal{X}$.

A $\mathcal{X}^+$-*tree* T is a finite rooted tree with edge labels from $\mathcal{X}^+$. For each $a \in \mathcal{X}$, every node v in T has at most one a-edge $v \xrightarrow{aw} v'$, for some v'. Let T be a $\mathcal{X}^+$-tree. A node in T is *branching* if it has at least two outgoing edges. A *leaf* in T is a node in T with no outgoing edges. An *inner node* in T is either the *root* or a node with at least one outgoing edge. $path(v)$ denotes the concatenation of the edge labels on the path from the *root* of T to the node v. Due to the requirement of unique a-edges at each node of T, paths are also unique. Therefore, we denote node v by $\overline{w}$ if and only if $w = path(v)$. The node $\overline{\varepsilon}$ is the *root*. Let $\overline{w}$ be a node in T. $|w|$ is the *depth* of $\overline{w}$. A sequence w *occurs* in T if T contains a node $\overline{wu}$, for some sequence u. $words(T)$ denotes the set of sequences occurring in T. A $\mathcal{X}^+$-tree is *atomic* if every edge is labeled by a *single* character from $\mathcal{X}$. A $\mathcal{X}^+$-tree is *compact* if every node is either the *root*, a leaf, or a branching node. An atomic as well as a compact $\mathcal{X}^+$-tree T is uniquely determined by $words(T)$.

2.2 Information Sources and Typical Sets

An *information* or *data source* is a random sequence $\{X_i\}$, where $-\infty < i < \infty$. We assume that the random sequence is stationary and ergodic, and X_i takes values from $\mathcal{X}$. The probability law defining the data source is given by

$$P_A(x_1^n) = Pr\{X_1^n = x_1^n\}, \qquad n \geq 1 \tag{1}$$

P_A is the actual probability of the data source. The *entropy* of the data source is

$$H \;=\; \lim_{n\to\infty} \sum_{x \in \mathcal{X}^n} \frac{1}{n} P_A(x) \log \frac{1}{P_A(x)} \;=\; \lim_{n\to\infty} \frac{1}{n} E \, \log \frac{1}{P_A(X_1^n)}.$$

The indicated limit always exists. It is easy to show that $H \leq \log k$, with equality if and only if the $\{X_i\}$ are i.i.d. and equally probable.

$$T(n, \epsilon) \;=\; \left\{ x \in \mathcal{X}^n : \left| \frac{1}{n} \log \frac{1}{P_A(x)} - H \right| \leq \epsilon \right\} \tag{2}$$

is called the set of *typical* sequences over $\mathcal{X}$. Obviously, $P_A(x) = 2^{-n(H \pm \epsilon)}$ for any $x \in T(n, \epsilon)$. The "Asymptotic Equipartition Property" (AEP for short) is a fundamental theorem in information theory. It is due to Shannon and McMillan (see [4]), and states that with high probability a sequence $x \in \mathcal{X}^n$ is typical, if $\epsilon > 0$ is fixed and n becomes large.

Theorem 1 For fixed $\epsilon > 0$, $\lim_{n \to \infty} Pr(T(n, \epsilon)) = 1$. $\Box$

$T(n, \epsilon)$ is not too large, as shown by the following proposition.

Proposition 1 $|T(n, \epsilon)| \leq 2^{n(H+\epsilon)}$

The following Shannon-McMillan-Breiman Theorem (cf. [4]) is a stronger version of Theorem 1:

Theorem 2 For a stationary ergodic source, with probability 1, as $n \to \infty$,

$$\frac{1}{n} \log \frac{1}{P_A(X_1^n)} \to H. \ \Box$$

Theorem 1 or 2 and Proposition 1 together imply that the set $T(n, \epsilon)$ has at most $2^{n(H+\epsilon)}$ members and for large n its probability approaches 1. This suggests the following ("fixed-to-variable-length") lossless code: Use one bit to mark any sequence $x \in \mathcal{X}^n$ as typical or not typical. If the source statistics is known, index all typical sequences and use the index of $x \in T(n, \epsilon)$ to encode x with at most $n(H + \epsilon)$ bits. Encode sequence $x \in \mathcal{X}^n \setminus T(n, \epsilon)$ using at most $\lceil n \log k \rceil$ bits.

2.3 Arithmetic Coding

In practice we cannot always index the members of $T(n, \epsilon)$, due to limitations in time and space. Therefore we have to introduce a sequential strategy. Assume that $\{X_i\}$ is i.i.d. and suppose P is a probability distribution on $\mathcal{X}$. Hence, for any $x \in \mathcal{X}^n$ we have

$$P_A(x) = P^n(x) = \prod_{i=1}^{n} P(x_i)$$

Suppose we can choose codewords of length $\lceil -\log P_A(x) \rceil$ for all $x \in \mathcal{X}^n$. Then we can upper bound the expected codeword length $\overline{L}_C(P^n)$ by $H(P^n) + 1$. In other words, the distance to the entropy is less than one bit.

Using a Huffman code C_h (for P) to encode $x \in \mathcal{X}^n$ by $C_h(x_1) \ldots C_h(x_n)$, we obtain

$$\overline{L}_{C_h}(P^n) < n(H(P) + 1) = H(P^n) + n.$$

That is, in the worst case, the distance of $\overline{L}_{C_h}(P^n)$ and the entropy can be arbitrary close to n. For this reason, we prefer to use arithmetic coding, a technique introduced by Elias (unpublished, but described by Abramson [1] and by Jelinek [12]). To explain arithmetic coding, we first have to introduce Shannon-Fano-Elias Coding.

2.3.1 Shannon-Fano-Elias Coding

Let P be a probability distribution on $\mathcal{X}$ such that $P(a) > 0$ for all $a \in \mathcal{X}$. We define two cumulative distribution functions f_P and $\overline{f_P}$ on $\mathcal{X}$ as follows:

$$f_P(a) = \sum_{b \in \mathcal{X}, b \leq a} P(b) \tag{3}$$

$$\overline{f_P}(a) = \sum_{b \in \mathcal{X}, b < a} P(b) + \frac{1}{2} P(a) \tag{4}$$

f_P divides $[0, 1]$ into disjoint subintervals. The subinterval whose right border is $f_P(a)$ is called the *interval for* a. $\overline{f_P}(a)$ is obviously the midpoint of the interval for a. Since all probabilities are strictly positive, we have $f_P(b) \neq f_P(c)$ for $b \neq c$. Hence a can be determined from $\overline{f_P}(a)$.

In general, $\overline{f_P}(a)$ is a real number which can only be described by an infinite number of bits. Therefore, $\overline{f_P}(a)$ must be rounded to be used as code for a. Let $L(a) = \left\lceil \log \frac{1}{P(a)} \right\rceil + 1$ and $\left\lfloor \overline{f_P}(a) \right\rfloor_{L(a)}$ be the number represented by the most significant $L(a)$ bits of the binary representation of $\overline{f_P}(a)$. We have

$$\overline{f_P}(a) - \left\lfloor \overline{f_P}(a) \right\rfloor_{L(a)} < \frac{1}{2^{L(a)}} \leq \frac{P(a)}{2} = \overline{f_P}(a) - f_P(a-1) \tag{5}$$

and therefore $\left\lfloor \overline{f_P}(a) \right\rfloor_{L(a)}$ is in the interval for a. Thus $L(a)$ bits are sufficient to encode a. The function $C_{\text{sfe}} : \mathcal{X} \to \mathbb{R}$ defined by $C_{\text{sfe}}(a) = \left\lfloor \overline{f_P}(a) \right\rfloor_{L(a)}$ is the *Shannon-Fano-Elias code.*

It is easy to see that after rounding the intervals remain disjoint. This implies that the Shannon-Fano-Elias code is prefix-free.

The expected length $\overline{L}_{C_{\text{sfe}}}(P)$ of the Shannon-Fano-Elias code is

$$\overline{L}_{C_{\text{sfe}}}(P) = \sum_{a \in \mathcal{X}} P(a) L(a) = \sum_{a \in \mathcal{X}} P(a) \left(\left\lceil \log \frac{1}{P(a)} \right\rceil + 1 \right) < H(P) + 2$$

That is, it is less than 2 bits larger than the entropy.

Example 1 Table 1 shows the construction of the Shannon-Fano-Elias code for the alphabet $\mathcal{X} = \{1, \ldots, 5\}$ with a given probability distribution. The distribution is not dyadic, so that for $a \in \{3, 4, 5\}$, $\overline{f_P}(a)$ does not have a finite representation. The resulting code is on average 1.2 bits longer than the corresponding Huffman code for this distribution.

The next section extends the concept of Shannon-Fano Elias coding.

a	$P(a)$	$f_P(a)$	$\overline{f_P}(a)$	$\overline{f_P}(a)$ in binary	$L(a)$	C_{sfe}
1	0.25	0.25	0.125	0.001	3	001
2	0.25	0.5	0.375	0.011	3	011
3	0.2	0.7	0.6	$0.1\overline{0011}$	4	1001
4	0.15	0.85	0.775	$0.110\overline{0011}$	4	1100
5	0.15	1.0	0.925	$0.111\overline{0110}$	4	1110

Table 1: Shannon-Fano-Elias code

2.3.2 The Arithmetic Coding Algorithm

The basic idea of the arithmetic coding algorithm is to represent a sequence $x \in \mathcal{X}^n$ by the midpoint of a subinterval of $[0, 1)$. Given a probability distribution P over $\mathcal{X}$, one processes x character by character from left to right. In each step one calculates boundaries of a subinterval according to P and the next symbol in x. The size of the subinterval becomes smaller and smaller, and the number of bits needed to specify the interval grows. Frequent symbols reduce the size of the subinterval by a smaller amount than rare symbols do and hence they add fewer bits to the code.

The algorithm computes a sequence of pairs (B_i, W_i), $i \in [0, n]$. B_i is the left border of a subinterval of $[0, 1)$ and W_i is the width of this subinterval. In each step, $[B_i, B_i + W_i)$ is divided into subintervals, one for each character in $\mathcal{X}$. x_{i+1} determines the next subinterval $[B_{i+1}, B_{i+1} + W_{i+1})$. The codeword $\widehat{x}$ for x is the midpoint $B_n + \frac{1}{2}W_n$ of the final subinterval $[B_n, B_n + W_n)$.

Algorithm 1
Input: $x \in \mathcal{X}^n$
Output: $\widehat{x}$
$B := 0$
$W := 1$
for $i := 1$ **to** n **do**
    $B := B + W \cdot f_P(x_i)$
    $W := W \cdot P(x_i)$
$\widehat{x} := B + \frac{1}{2}W$

Example 2 Consider the alphabet $\mathcal{X} = \{\texttt{a},\texttt{b},\texttt{c},\texttt{d}\}$ and the probability distribution in Table 2. For $x = \texttt{aabc}$ Algorithm 1 computes the following sequence of intervals whose borders are given in binary representation:

$$[0, 1) \xrightarrow{\texttt{a}} [0, 0.1) \xrightarrow{\texttt{a}} [0, 0.01) \xrightarrow{\texttt{b}} [0.001, 0.0011) \xrightarrow{\texttt{c}} [0.0010110, 0.0010111)$$

Thus $\widehat{x} = 00101101$.

a	$P(a)$	$P(a)$ in binary	$\overline{f_P}(a)$
`a`	0.5	.100	.000
`b`	0.25	.010	.100
`c`	0.125	.001	.110
`d`	0.125	.001	.111

Table 2: A probability distribution for the alphabet $\{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$

Decoding $\widehat{x}$ is very similar to encoding: One starts with the interval $[B_0, B_0 + W_0) := [0, 1)$ and divides this interval into subintervals according to P. One determines the shortest prefix of $\widehat{x}$ which uniquely determines one of these subintervals. The subinterval in turn determines x_1. One iterates these steps until $\widehat{x}$ is completely read. Then x is decoded.

Note that in practice the direct application of Algorithm 1 (and the corresponding decoding algorithm) leads to precision problems. Witten, Neal and Cleary [28] have shown how to circumvent these problems.

2.3.3 Coding Redundancy

Suppose encoder and decoder both have access to a distribution P_C such that $P_C(x_1^t) > 0$ for all $x_1^t \in \mathcal{X}^t$, $t \in [0, n]$. We call P_C the *coding distribution* and require that it satisfies the following properties for any $t \in [0, n]$:

$$\begin{aligned} P_C(\varepsilon) &= 1 \\ P_C(x_1^{t-1}) &= \sum_{a \in \mathcal{X}} P_C(x_1^{t-1}, X_t = a) \text{ for all } x_1^{t-1} \in \mathcal{X}^{t-1} \end{aligned}$$

Besides the coding distribution we also have an actual distribution P_A defined by the source. This is possibly different from the coding distribution. That is, the coding distribution is the best estimate we can achieve, if the actual distribution is not known.

Since $L(a) = \left\lceil \log \frac{1}{P(a)} \right\rceil + 1$ for any $a \in \mathcal{X}$, we obtain the following theorem:

Theorem 3 [4, 27] Let $L(x) = |\widehat{x}|$ for any $x \in \mathcal{X}^n$. For any coding distribution P_C, we have

$$L(x) < \log \frac{1}{P_C(x)} + 2. \qquad (6)$$

The codewords form a prefix code. □

The difference between the codeword length $L(x)$ and $\log \frac{1}{P_C(x)}$ is always less than 2 bits. We say that the individual *coding redundancy* is less than 2 bits.

2.4 Unknown Parameters

In the previous sections we always assumed that a particular probability distribution over $\mathcal{X}$ was given. In practice, however, we do not know the distribution. We now consider how to estimate it. For ease of presentation we restrict ourselves to the binary alphabet $\mathcal{B} = \{0, 1\}$.

Let $\theta \in [0, 1]$. The probability that a memoryless source with parameter θ generates a sequence $x \in \mathcal{B}^n$ with l zeros and r ones is $P_A(l, r) = (1-\theta)^l \theta^r$. If we weight this probability over θ with a $(\frac{1}{2}, \frac{1}{2})$-Dirichlet distribution, then we obtain the Krichevsky-Trofimov estimator.

Definition 1 [13] The *Krichevsky-Trofimov estimated probability*, KT-estimator for short, for a sequence containing $l \geq 0$ zeros and $r \geq 0$ ones is defined as

$$P_{KT}(l, r) = \int_0^1 \frac{1}{\pi\sqrt{(1-\theta)\theta}} (1-\theta)^l \theta^r \, d\theta$$

The sequential behavior of the KT-estimator was studied by Shtarkov:

Lemma 1 [22]

1. $P_{KT}(l, r)$ can be computed sequentially by the following equations:

$$\begin{aligned} P_{KT}(0,0) &= 1 \\ P_{KT}(l+1, r) &= \frac{l + \frac{1}{2}}{l + r + 1} \cdot P_{KT}(l, r) \text{ for } l, r \geq 0 \\ P_{KT}(l, r+1) &= \frac{r + \frac{1}{2}}{l + r + 1} \cdot P_{KT}(l, r) \text{ for } l, r \geq 0 \end{aligned}$$

2. For $l + r \geq 1$, $P_{KT}(l, r) \;\geq\; \frac{1}{2} \frac{1}{\sqrt{l+r}} \left(\frac{l}{l+r}\right)^l \left(\frac{r}{l+r}\right)^r$.

3. For all $l + r \geq 1$ and all $\theta \in [0, 1]$,

$$\log \frac{(1-\theta)^l \theta^r}{P_{KT}(l, r)} \;\leq\; \log \frac{(1-\theta)^l \theta^r}{\frac{1}{2} \frac{1}{\sqrt{l+r}} \left(\frac{l}{l+r}\right)^l \left(\frac{r}{l+r}\right)^r} \;\leq\; \frac{1}{2} \log(l+r) + 1. \; \square$$

Note that the upper bound in Statement 3 does not depend on the parameter θ. We say that the parameter redundancy is uniformly bounded.

Now we consider the effect of the KT-estimator on the coding redundancy:

Lemma 2 Suppose we use the KT-estimator in arithmetic coding. Then for any $x \in \mathcal{B}^n$,

$$L(x) < -\log P_A(x) + \frac{1}{2} \log n + 3. \tag{7}$$

Proof: We first note that P_{KT} is a coding distribution. Using arithmetic coding with the KT-estimator, we obtain $L(x) < \log \frac{1}{P_{KT}(x)} + 2$, according to Theorem 3. Since $P_A(x) > 0$, we conclude

$$\begin{aligned} L(x) &< \log \frac{1}{P_{KT}(x)} - \log P_A(x) + \log P_A(x) + 2 \\ &= -\log P_A(x) + \log \frac{P_A(x)}{P_{KT}(x)} + 2. \end{aligned}$$

Since x is binary, we can apply Statement 3 of Lemma 1 to upper bound $\log \frac{P_A(x)}{P_{KT}(x)}$ by $\frac{1}{2} \log n + 1$. This implies the claim. □

If the actual distribution is known, then $L(x) < \log \frac{1}{P_A(x)} + 2$, according to Theorem 3. If the actual distribution is not known, then we can use the KT-estimator instead. This incurs a penalty of $\frac{1}{2} \log n + 1$ bits.

The KT-estimator and its properties can be generalized to arbitrary alphabets [22,23]. Note that for an alphabet of size k there are $k-1$ unknown parameters.

2.5 Context Trees

The results of the previous sections are restricted to i.i.d. sources. In this section we consider a certain class of sources which are *not* i.i.d. We show how to apply the previous results to these. The idea is to employ a model that gives the probability for the next symbol, depending on it's context. An entropy encoder can use this probability to form a codeword. Rissanen [20] showed that a context based method allows for a good prediction of the probability of the next symbol.

Definition 2 A *context tree* CT is an atomic $\mathcal{X}^+$-tree s.t. each inner node has k outgoing edges. Each leaf $\overline{c}$ is labeled by a probability distribution $P_{CT}(\,.\mid c^{-1})$. For ease of notation we identify a leaf $\overline{c}$ and the sequence c. A source is a *tree source* if and only if there is a context tree CT such that for any $x \in \mathcal{X}^n$ we have

$$P_A(x) = P_A(x_1 \dots x_{l(x)}) \prod_{i=l(x)+1}^{n} P_{CT}(x_i \mid (c_i)^{-1}) \tag{8}$$

where

- $l(x)$ is the smallest integer $i \in [1, n]$ such that $(x_1^i)^{-1}$ is a leaf in CT.
- For any $i \in [l(x)+1, n]$, c_i is a leaf in CT such that $(c_i)^{-1} = x_{i-|c_i|}^{i-1}$. c_i is called the *context* of x_i in x w.r.t. CT.

A context tree CT satisfying (8) for any $x \in \mathcal{X}^n$, is called *model of the tree source.*

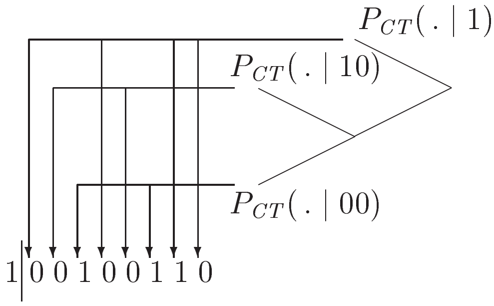

Figure 1: A context tree and the contexts for a given sequence

Example 3 Consider the binary context tree in Figure 1 with the leafs 00, 01 and 1. Suppose $P_{CT}(1 \mid 00) = 0.5$, $P_{CT}(1 \mid 10) = 0.3$, $P_{CT}(1 \mid 1) = 0.1$, and let $x = 100100110$. Then $n = 9$, $l(x) = 1$, and

$$\begin{aligned}\prod_{i=l(x)+1}^{n} P_{CT}(x_i \mid (c_i)^{-1}) &= P_{CT}(0 \mid 1) \cdot P_{CT}(0 \mid 10) \cdot P_{CT}(1 \mid 00) \cdot P_{CT}(0 \mid 1) \cdot \\ &\quad P_{CT}(0 \mid 10) \cdot P_{CT}(1 \mid 00) \cdot P_{CT}(1 \mid 1) \cdot P_{CT}(0 \mid 1) \\ &= 0.9 \cdot 0.7 \cdot 0.5 \cdot 0.9 \cdot 0.7 \cdot 0.5 \cdot 0.1 \cdot 0.9 \\ &= 0.00893025.\end{aligned}$$

Suppose that the source is a tree source modeled by a context tree CT. P_{CT} does only depend on a character and its context. It does not depend on where the character or the context occurs. Hence we have

$$\prod_{i=l(x)+1}^{n} P_{CT}(x_i \mid (c_i)^{-1}) \;=\; \prod_{c \in \mathcal{L}(CT)} \;\prod_{i \in Sub(x,c)} P_{CT}(x_i \mid c^{-1})$$

where $\mathcal{L}(CT)$ is the set of leafs in CT and $Sub(x,c) = \{i \in [l(x)+1, n] : c^{-1} = x_{i-|c|}^{i-1}\}$ for any $c \in \mathcal{L}(CT)$. We have $[l(x)+1, n] = \bigcup_{c \in \mathcal{L}(CT)} Sub(x,c)$. Moreover, for each $c \in \mathcal{L}(CT)$, the subsequence $\{X_i\}_{i \in Sub(x,c)}$ is i.i.d. Thus the corresponding subsequence of x can be encoded from left to right as described in the previous sections. But notice that there are extra costs if the model is not known.

Remark 1 Using the context tree weighting technique (cf. [27]) the length $L(x)$ of the codeword for a binary sequence $x \in \mathcal{B}^n$ w.r.t. the actual probability distribution can be upper bounded by

$$L(x) < -\log P_A(x) + 2l + \frac{l}{2} \log \frac{n}{l} + 1, \tag{9}$$

where l is the number of leafs in a context tree modeling the source.

3 The Burrows and Wheeler-Transformation

In this section we define the Burrows and Wheeler-Transformation [3] and study its properties. We show how to construct it and how to decode it in linear time and space. The idea of the Burrows and Wheeler-Transformation is to permute the characters of the input sequence in such a way that characters with the same *right* context are grouped together. Note that most other compression schemes consider the *left* contexts of the characters in the input sequence.

We assume that $x \in \mathcal{X}^*$ is a sequence of length $n \geq 1$ and $\$ \in \mathcal{X}$ is a character not occurring in x, the *sentinel* character. We furthermore suppose that \$ is the maximal character in $\mathcal{X}$. For any $i \in [1, n+1]$, let $S_x(i) = x_i \dots x_n\$$ denote the *ith* non-empty suffix of $x\$$. Note that due to the sentinel no $S_x(i)$ is a proper prefix of any $S_x(j)$. Let $S_x(j_1), S_x(j_2), \dots, S_x(j_{n+1})$ be the sequence of all non-empty suffixes of $x\$$ in lexicographic order. This gives a bijective mapping $\varphi_x : [1, n+1] \to [1, n+1]$ defined by $\varphi_x(i) = j_i$. φ_x is the *suffix order on* $x\$$. Note that $\varphi_x(n+1) = n+1$, since $S_x(n+1) = \$$ is the maximal character in $\mathcal{X}$. For convenience, we sometimes write φ_x as a list $\varphi_x(1), \varphi_x(2), \dots, \varphi_x(n+1)$.

The *Burrows and Wheeler-Transformation* of x is the sequence $\widetilde{x}$ of length $n+1$ such that for any $i \in [1, n+1]$:

$$\widetilde{x}_i = \begin{cases} \$ & \text{if } \varphi_x(i) = 1 \\ x_{\varphi_x(i)-1} & \text{otherwise} \end{cases}$$

Note that Burrows and Wheeler [3] define their transformation by considering all cyclic shifts of x. If one sorts these lexicographically and writes them line by line beginning with the smallest one, then the last column of the resulting matrix is the original transformation. Burrows and Wheeler later also appended a sentinel character to x, recognizing the fact that it is more efficient to sort non-empty suffixes than cyclic shifts. However, there is another reason for appending the sentinel: it prevents from introducing dependencies between parts of the input sequence which are actually not present. For these two reasons, we have given a modified definition of the Burrows and Wheeler-Transformation.

Example 4 Let $\mathcal{X} = \{a, b\}$ and $x = abab$. The following table shows the non-empty suffixes of $x\$$ in lexicographic order and the Burrows and Wheeler-Transformation of x:

	ordered suffixes					$\widetilde{x}$
$S_x(1)$	a	b	a	b	\$	\$
$S_x(3)$	a	b	\$			b
$S_x(2)$	b	a	b	\$		a
$S_x(4)$	b	\$				a
$S_x(5)$	\$					b

Thus $\widetilde{x} = \$baab$. To obtain the Burrows and Wheeler-Transformation according to the definition in [3], one sorts the cyclic shifts of $abab$, to obtain the transformation $bbaa$:

a	b	a	b
a	b	a	b
b	a	b	a
b	a	b	a

The original Burrows and Wheeler-Transformation results in a sequence of length n, while our transformation leads to a sequence of length $n+1$. This is because we include the sentinel to mark the position corresponding to the longest suffix $S_x(1)$. Burrows and Wheeler instead use an extra integer to store that position.[1]

3.1 Properties

In this section we show that the Burrows and Wheeler-Transformation permutes a tree source in such a way, that the permutation can be partitioned into intervals, one for each leaf of the context tree. In each such interval the subsequence of the tree source is i.i.d.

Theorem 4 Suppose that the source is a tree source with a model CT. Let $r = |\mathcal{L}(CT)|$, and $c_1, \ldots, c_r$ be the leafs of CT in lexicographic order. Let $x \in \mathcal{X}^n$ be generated by the source, and define $y = x^{-1}$. Let z be obtained from $\widetilde{y}$ by deleting the sentinel in $\widetilde{y}$ and the characters at all positions $i \in [1, n+1]$ with $\varphi_y(i) \geq n + 2 - l(x)$. Then there are sequences $w_1, \ldots, w_r$ such that

1. $z = w_1 \ldots w_r$.
2. Let $j \in [1, r]$, $l_j = |Sub(x, c_j)|$, and $Sub(x, c_j) = \{i_1, i_2, \ldots, i_{l_j}\}$ such that $S_y(n{+}2{-}i_1), S_y(n{+}2{-}i_2), \ldots, S_y(n{+}2{-}i_{l_j})$ are in lexicographic order. Then $w_j = x_{i_1} x_{i_2} \ldots x_{i_{l_j}}$ and the subsequence of the tree source corresponding to w_j is i.i.d.

Proof: By Definition 2, the first $l(x)$ characters in x do not have a context w.r.t. CT. On the other hand, for any $i \in [l(x)+1, n]$, x_i has a context in x w.r.t. CT. Thus for any $i \in [l(x)+1, n]$ there is a leaf c in CT such that c is a prefix of the suffix $y_{n+2-i} y_{n+2-i+1} \ldots y_n$ of y. Append the sentinel to each of these suffixes. This gives the set $\{S_y(n+2-i) \mid i \in [l(x)+1, n]\}$ of non-empty suffixes of $y\$$. Consider these suffixes in lexicographic order. They correspond to the elements in $\widetilde{y}$ which are also present in z. Due to the lexicographic order, all suffixes with the same prefix are grouped together. Partition the ordered sequence of suffixes into factors, such that each factor consists of all suffixes having the same leaf c

[1] However, when implementing $\widetilde{x}$ we also use an extra integer, see Section 4.3.

of the context tree as a prefix. This also partitions z into factors $w_1, \ldots, w_r$ such that $z = w_1 \ldots w_r$. Note that w_j is the empty sequence, if c_j is a leaf in CT but there is no i such that c_j is a context of x_i in x w.r.t. CT. For each $q \in [1, l_j]$, c_j is the context of $y_{(n+2-i_q)-1} = y_{n+1-i_q} = x_{i_q}$ in x w.r.t. CT. Hence c_j is a prefix of $S_y(n+2-i_q)$, and thus $w_j = y_{(n+2-i_1)-1} y_{(n+2-i_2)-1} \cdots y_{(n+2-i_{l_j})-1} = y_{n+1-i_1} y_{n+1-i_2} \cdots y_{n+1-i_{l_j}} = x_{i_1} x_{i_2} \ldots x_{i_{l_j}}$. Due to the properties of context trees, it is clear that the subsequence $\{X_i\}_{i \in Sub(x,c_j)}$ is i.i.d. □

Note that we can evaluate $P_A(x)$ by considering the leafs of CT in lexicographic order, that is

$$P_A(x) = P_A(x_1 \ldots x_{l(x)}) \cdot \prod_{j=1}^{r} \prod_{i \in Sub(x,c_j)} P_{CT}(x_i \mid (c_j)^{-1})$$

Example 5 Let $x = 100100110$ and consider the context tree of Example 3. Then $n = 9$, $l(x) = 1$, $Sub(x, 00) = \{7, 4\}$, $Sub(x, 01) = \{6, 3\}$ and $Sub(x, 1) = \{8, 5, 9, 2\}$. We have $y = x^{-1} = 011001001$ and the following suffix order φ_y (the reverse contexts are shown in bold face):

ordered suffixes										$\widetilde{y}$
0	**0**	1	0	0	1	\$				1
0	**0**	1	\$							1
0	**1**	0	0	1	\$					0
0	1	1	0	0	1	0	0	1	\$	\$
0	**1**	\$								0
1	0	0	1	0	0	1	\$			1
1	0	0	1	\$						0
1	1	0	0	1	0	0	1	\$		0
1	\$									0
\$										1

Thus $\widetilde{y} = 110\$010001$. To obtain $z = 11001000$, we delete \$ in $\widetilde{y}$ and the suffix of length 1. Now $z = w_1 w_2 w_3$ where $w_1 = x_7 x_4$, $w_2 = x_6 x_3$, and $w_3 = x_8 x_5 x_9 x_2$.

If the model CT of the tree source is known, then one can split the sequence z (see Theorem 4) into factors $w_1, \ldots, w_r$ and encode each w_j as described in the previous sections. In this way it is possible to achieve the same compression rate as a method which does without the Burrows and Wheeler-Transformation. However, such a method requires to store at each leaf of CT a statistic. In contrast, after the Burrows and Wheeler-Transformation one only needs one statistic at any time: when encoding w_j one only needs to store the statistic for the context c_j. Thus the space consumption is smaller by a factor $|\mathcal{L}(CT)|$. Another important advantage of applying the Burrows and Wheeler-Transformation is that it allows to handle contexts of arbitrary length, while a method which does without has to restrict the depth of the context tree and thus the length the contexts, due to space limitations in practice.

Unfortunately, if we do not know the model of the tree source, then we also do not know when to change from one context to another. In Section 4.6, we will show how to tackle this problem.

3.2 Linear Time Construction

The construction of the Burrows and Wheeler-Transformation accounts for most of the resources required by a data compression program based on this transformation. Therefore, we carefully consider construction methods.

In order to construct the Burrows and Wheeler-Transformation $\widetilde{x}$, one first computes the suffix order on $x\$$. In [3] it was observed that this can be done in linear time and space, using the suffix tree for x. In our opinion, suffix trees provide the method of choice for computing the suffix order, from a theoretical as well as a practical point of view:

- There are methods to construct the suffix tree for x in $O(n)$ space and $O(kn)$ time [7,18,25,26]. The suffix tree can be organized, such that a simple depth first traversal (in linear time and space) gives the suffix order on $x\$$. These complexities are for the worst case. Thus a suffix tree based method has a predictable running time. This is not true for the methods described in [3] whose running time is $O(n^2 \log n)$ in the worst case (although they are fast in the average case). We refer to these as *non-linear methods*.
- In [14] it was recently shown that the suffix tree for x can be computed in $O(kn)$ time using only $12n$ bytes of space in the worst case, and about $10n$ bytes in the average case. The space consumption of a suffix tree based method is thus comparable to the non-linear methods which require about $8n$ bytes of space.
- A careful program design leads to a suffix tree based method which runs fast in practice.[2]

In the following we will briefly introduce suffix trees and describe how they can be used to compute $\widetilde{x}$.

The *suffix tree* for x, denoted by ST, is the compact $\mathcal{X}^+$-tree T such that $words(T) = \{w \in \mathcal{X}^* \mid w \text{ is a factor of } x\$\}$. Figure 2 shows the suffix tree for the sequence $x = abab$. Due to the sentinel character, there is a one-to-one correspondence between the leafs of ST and the non-empty suffixes of $x\$$: each

[2] Burrows and Wheeler [3] report that they have implemented a suffix tree based method to compute the Burrows and Wheeler-Transformation. However, they do not give enough information to substantially evaluate how their implementation performs in comparison to the non-linear methods.

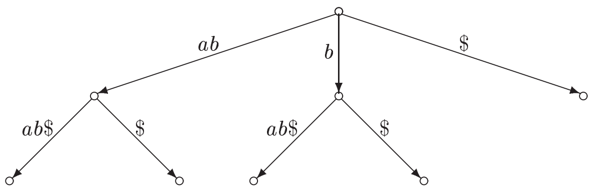

Figure 2: The suffix tree for $x = abab$

suffix $S_x(i)$ is represented by the leaf $\overline{S_x(i)}$, and different leafs represent different suffixes. This implies that ST has exactly $n+1$ leaves. Moreover, since $n \geq 1$ and $x_1 \neq \$$, the *root* of ST is branching. Hence each inner node in ST is branching. This means that there are at most n inner nodes in ST. Each node can be represented in constant space. Since ST has at most $2n+1$ nodes, the number of edges is bounded by $2n$. Each edge is labeled by a factor of $x\$$. Such a label can be represented in constant space by a pair of pointers into $x\$$. Hence ST can be represented in $O(n)$ space.

Due to the one-to-one correspondence of the leafs of ST and the non-empty suffixes of $x\$$, the Burrows and Wheeler-Transformation can be read from ST by a simple depth first traversal. This processes the edges outgoing from some branching node $\overline{w}$ in order $<_{\overline{w}}$ which is defined as follows:

$$\overline{wau} \xrightarrow{au} \overline{wau} \quad <_{\overline{w}} \quad \overline{wcv} \xrightarrow{cv} \overline{wcv} \iff a < c$$

Since no two edges outgoing from $\overline{w}$ have a label beginning with the same character, $<_{\overline{w}}$ is total. It is obvious that such a depth first traversal visits leaf $\overline{S_x(i)}$ before leaf $\overline{S_x(j)}$ if and only if $S_x(i) < S_x(j)$. Thus the suffix order $\varphi_x(1), \varphi_x(2), \ldots, \varphi_x(n+1)$ on $x\$$ is just the list of suffix numbers encountered at the leafs during the traversal. If one implements the suffix tree in such a way that the edges outgoing from a branching node $\overline{w}$ are ordered by $<_{\overline{w}}$, then the depth first traversal runs in linear time. No extra space is needed, except for the output sequence $\widetilde{x}$.

Example 6 Consider the suffix tree ST for $x = abab$ in Figure 2. The depth first traversal visits the nodes of ST in the order *root*, $\overline{ab}$, $\overline{S_x(1)}$, $\overline{S_x(3)}$, $\overline{b}$, $\overline{S_x(2)}$, $\overline{S_x(4)}$, $\overline{S_x(5)}$. This gives the suffix order $\varphi_x = 1, 3, 2, 4, 5$ on $x\$$ (see also Example 4).

Linear time suffix constructions have a long history, starting with the construction of Weiner [26]. Later authors [18, 25] have developed improved algorithms. Giegerich and Kurtz [10] reveal that these three linear time algorithms are very closely related, although they are all based on rather different intuitive ideas. Recently Farach [7] described a linear time algorithm, which differs very much from the other algorithms.

For our particular application, we consider McCreight's algorithm [18] to be the best choice. This is for the following reasons: At first, we do not need the additional virtue of Ukkonen's algorithm (it is online) and of Farach's algorithm (it can handle integer alphabets). Second, McCreight's algorithm is more space efficient than Weiner's algorithm, and slightly faster than Ukkonen's method, as shown in [9]. We have not seen any practical results of the space and time behavior of Farach's algorithms. We note that McCreight's algorithm also requires the sentinel character appended to the input sequence x. Thus it is well suited for computing the Burrows and Wheeler-Transformation.

3.3 Decoding

Since our definition of the Burrows and Wheeler-Transformation slightly differs from the original, we now present an algorithm to decode x given $\widetilde{x}$. The algorithm runs in $O(n)$ time and space and is divided into three phases. It is similar to the algorithm given in [3].

In the first phase of the decoding algorithm, two tables $count : \mathcal{X} \to [0,n]$ and $\mathit{offset} : [1, n+1] \to [1, n]$ are computed. They are specified as follows:

- For any $a \in \mathcal{X}$, $count[a]$ is the number of occurrences of a in $x\$$.
- For any $r \in [1, n+1]$, $\mathit{offset}[r]$ is the number of positions $p \in [1, r]$ such that $\widetilde{x}_p = \widetilde{x}_r$. That is, $\mathit{offset}[r] = l$ if and only if position r is the lth position in $\widetilde{x}$ (from left to right) where character $\widetilde{x}_r$ occurs.

Note that $\widetilde{x}$ is just a permutation of $x\$$. Hence *count* can be computed in one pass over $\widetilde{x}$. In the same pass, one can also compute *offset*.

In the second phase, a table $base : \mathcal{X} \to [0, n]$ is computed such that for any $a \in \mathcal{X}$

$$base[a] = \sum_{b \in \mathcal{X}, b < a} count[b]$$

That is, if $l = base[a]+1$, then the smallest non-empty suffix of $x\$$ beginning with character a is the lth smallest non-empty suffix of $x\$$. Note that $count[\$] = 1$ and $base[\$] = n$. Obviously, *base* can be computed in $O(k)$ time from *count*.

Example 7 For $x = abab$ we have $\widetilde{x} = \$baab$ (see Example 4). To decode x from $\widetilde{x}$, the tables *count*, *base*, and *offset* are computed, where $count[a] = count[b] = base[b] = 2$, $base[a] = 0$, and

r	1	2	3	4	5
$\mathit{offset}[r]$	1	1	1	2	2

In the third phase, x is decoded from right to left by computing for any $i \in [2, n+1]$ an index r_i with the property $\varphi_x(r_i) = i$. That is, suffix $S_x(i)$ is the $r_i th$ smallest non-empty suffix of $x\$$. Now suppose that $i \in [1, n]$ and r_{i+1} is given. Then $\varphi_x(r_{i+1}) = i + 1 \neq 1$ and therefore x_i can be computed from r_{i+1} and $\widetilde{x}$, due to the following property:

$$x_i = x_{i+1-1} = x_{\varphi_x(r_{i+1})-1} = \widetilde{x}_{r_{i+1}} \tag{10}$$

The following lemma shows how to compute r_i from x_i and r_{i+1}:

Lemma 3 For any $i \in [1, n+1]$ the following properties hold:

$$r_i = \begin{cases} n+1 & \text{if } i = n+1 \\ \mathit{base}[x_i] + \mathit{offset}[r_{i+1}] & \text{otherwise} \end{cases}$$

Proof: Since $\varphi_x(n+1) = n+1$ (see remark above), we have $r_{n+1} = n+1$. Now let $i \in [1, n]$ and $a = x_i$. Note that $a = \widetilde{x}_{r_{i+1}} \neq \$$. One easily observes that $\mathit{base}[a] + 1 \leq r_i \leq \mathit{base}[a] + \mathit{count}[a]$. If $S_x(i)$ is the only suffix beginning with a, then $\mathit{count}[a] = \mathit{offset}[r_{i+1}] = 1$. Hence $r_i = \mathit{base}[a] + 1 = \mathit{base}[a] + \mathit{offset}[r_{i+1}]$. Now suppose there is a suffix $S_x(i')$, $i' \in [1, n]$, $i' \neq i$, which also begins with a. Then $a = x_{i'} = \widetilde{x}_{r_{i'+1}}$. Moreover, the following equivalences holds:

$$S_x(i) < S_x(i') \iff S_x(i+1) < S_x(i'+1) \iff \mathit{offset}[r_{i+1}] < \mathit{offset}[r_{i'+1}]$$

where $<$ denotes the lexicographic order on $\mathcal{X}^*$. Hence, if $\mathit{offset}[r_{i+1}] = l$, then $S_x(i)$ is the lth suffix beginning with a. This implies $r_i = \mathit{base}[a] + \mathit{offset}[r_{i+1}]$. □

With Property (10) and Lemma 3 it is easy to show that the following algorithm correctly decodes x from $\widetilde{x}$ in $O(n)$ time and space.

Algorithm 2

Input: $\widetilde{x}$
Output: x

for all $a \in \mathcal{X}$ **do** $\mathit{count}[a] := 0$
for $i := 1$ **to** $n+1$ **do**
    $a := \widetilde{x}_i$
    $\mathit{count}[a] := \mathit{count}[a] + 1$
    $\mathit{offset}[i] := \mathit{count}[a]$
$\mathit{base}[1] := 0$
for $a := 2$ **to** k **do**
    $\mathit{base}[a] := \mathit{base}[a-1] + \mathit{count}[a-1]$
$r := n+1$
for $i := n$ **downto** 1 **do**
    $x_i := \widetilde{x}_r$
    $r := \mathit{base}[x_i] + \mathit{offset}[r]$

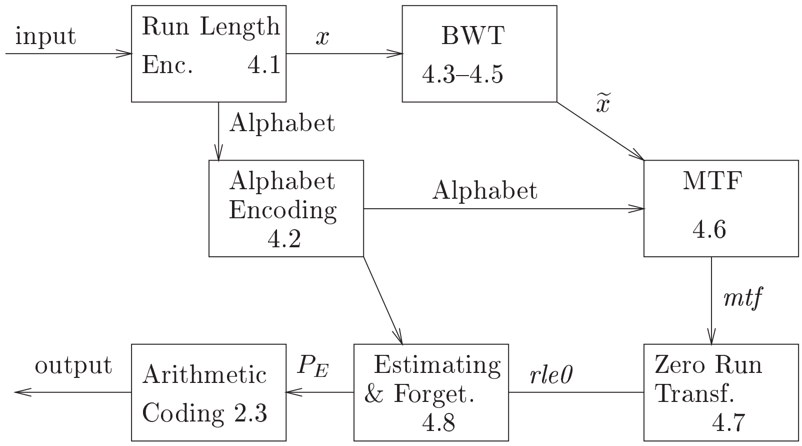

Figure 3: The data flow in our compression program

The algorithm needs $n + 1$ integers for table *offset*, $2k$ integers for tables *count* and *base*, and $2n + 1$ characters to store the input $\widetilde{x}$ and the output x. One can reuse the space for table *base* when computing the partial sums in *count*. This saves k integers. If an integer can be stored in 4 bytes and a character in 1 byte, then the space consumption for the decoding is, up to some additive constants, $4(n + k) + 2n = 6n + 4k$ bytes. In practice this can be reduced to $5n + 4k$ bytes.

4 Implementation Techniques

This section is devoted to the practical or engineering aspects of data compression using the Burrows and Wheeler-Transformation. We describe techniques to efficiently implement the previously discussed algorithms and data structures, and to make them work well in practice. We will always motivate why we chose the particular technique. The structure of this section follows the data flow of our data compression program, as depicted in Figure 3.

From now on, we assume that characters in the input sequence can be represented by one byte. That is, $\mathcal{X} \setminus \{\$\}$ is restricted to be a subset of the 256 character ASCII alphabet. Of course, we use the predefined order on this alphabet to sort characters, suffixes, and edges. We furthermore suppose that integers are represented by 4 bytes, i.e. 32 bits.

When we discussed the properties of the Burrows and Wheeler-Transformation in Section 3.1, we reversed the input sequence and then applied the transformation. In this way we were able to conveniently describe it's properties. From a practical point of view it is not necessary to reverse the input sequence. For this reason, our compression program directly computes the Burrows and Wheeler-Transformation for the input sequence, possibly after applying the run length encoding.

4.1 Run Length Encoding

A *run* in x is a non-empty factor of x which does not contain different characters. We have implemented a simple scheme which encodes a run of length $r > 3$ by $3 + \lfloor r/256 \rfloor$ characters: The first character marks that a run starts at its position, the second character is the character the run consists of, and the next $1 + \lfloor r/256 \rfloor$ characters add up to r, if they are interpreted as one-byte integers in the range $[0, 255]$. $rle(x)$ denotes the sequence obtained by applying the run length encoding to x.

In general, it is not a good idea to apply the run length encoding, since it disguises character dependencies. However, there are cases where it definitely should be applied: Suppose that x contains many runs. If we apply the move-to-front transformation (see Section 4.6) to $\widetilde{x}$, we obtain a sequence with up to 90% zeros. If we instead first apply the run length encoding to x, then this reduces the number of zeros to about 60%, as illustrated in Figure 4 for the file *pic* of the Calgary Corpus. In the latter case we can achieve much better estimates for the non-zero symbols, which in turn improves the compression rate. We consider x to contain many runs, if $|rle(x)| < 0.7n$. This threshold proved to be sensible in practice.

Thus we apply run length encoding if (i) there is an ASCII character available which does not appear in x (this is used for marking the start of a run), and (ii) if $|rle(x)| < 0.7n$ holds. In order to decide (i) and (ii), we first compute the set of characters actually occurring in x (we need this anyway, see Section 4.2), and determine the length of $rle(x)$. This can be done in one pass over x in linear time. In case we apply the run length encoding, we no longer need x later. So we can compute the encoding in place, using the space where x is stored. Thus our scheme runs in linear time without using extra space.

4.2 Alphabet Encoding

Most compression programs do not encode the set of characters which actually occur in the sequence to be compressed. This implies that one has to deal with the entire ASCII alphabet. For our approach this would mean that (i) we have more free parameters for our estimator (which enlarges coding redundancy), and (ii) we have to reserve at least one extra codeword for the set of symbols not occurring in x. Thus one of the codewords for those characters which actually occur in x has at least one extra bit. This would also lead to additional redundancy.

For these reasons, we do encode the alphabet. We have developed an alphabet encoding technique, which exploits that an alphabet usually consists of several intervals, i.e. sequences of at least two consecutive characters of the ASCII alphabet. For the alphabet encoding we need a function *binencode* which searches

i	codeword	i	codeword	i	codeword	i	codeword
1	00	4	1000	7	1011	10	1110
2	010	5	1001	8	1100	11	1111
3	011	6	1010	9	1101		

Table 3: Codewords returned by *binencode* for $l = 1$, $r = 11$, and $i \in [l, r]$.

for a number i in some interval $[l, r]$ using a binary strategy. In each step, $[l, r]$ is divided into two disjoint subintervals, and it is output whether i occurs in the first or the second subinterval. The function computes a codeword whose length is increasing with i. For each $l, r \in \mathbb{N}$, $r \geq l$, and each $i \in [l, r]$, *binencode* is specified as follows:

$$binencode(i, l, r) = \begin{cases} \varepsilon & \text{if } l = r \\ 0 \cdot binencode(i, l, r - j) & \text{else if } i \leq r - j \\ 1 \cdot binencode(i, r - j + 1, r) & \text{otherwise} \end{cases}$$
$$\text{where } j = \max_{q \in \mathbb{N}_0} \{2^q : 2^q < r - l + 1\}$$

Table 3 shows some codewords returned by *binencode*.

Suppose the alphabet is given as a sequence of one-byte integers $0 \leq a_1 < \ldots < a_k \leq 255$. In a first step, we reverse this sequence and rename each character, i.e. we compute $a'_1, \ldots, a'_k, a'_{k+1}$ where $a'_i = 255 - a_{k+1-i}$ for $i \in [1, k]$, and $a'_{k+1} = 256$. We obviously have $0 \leq a'_1 < \cdots < a'_{k+1}$. Let $b = 0$ and $l = 1$. Now we proceed as follows, until $b \geq 256$.

(1) If $a'_l + 1 = a'_{l+1}$, then let $j \in [l+1, k+1]$ be maximal such that $a'_i + 1 = a'_{i+1}$ for all $i \in [l, j-1]$. That is, $[a'_l, a'_j]$ is an interval of length $j - l + 1 \geq 2$. We encode the first symbol a'_l of the interval by the even number $2(a'_l - b) + 2$, and its length $j - l + 1$ by the number $j - l$. More precisely, we use the two codewords $binencode(2(a'_l - b) + 2, 1, 2(256 - b) + 1)$ and $binencode(j - l, 1, 256 - a'_l)$. We proceed with $b = a'_j + 2$ and $l = j + 1$.

(2) If $a'_l + 1 \neq a'_{l+1}$, then the single character a'_l is not part of an interval. It is encoded by the odd number $2(a'_l - b) + 1$, i.e. we use the codeword $binencode(2(a'_l - b) + 1, 1, 2(256 - b) + 1)$. We proceed with $b = a'_l + 2$ and $l = l + 1$.

Note that while b grows, the interval used for *binencode*, becomes smaller. The alphabet is encoded before x (see also Figure 3). Hence the estimated probability distribution is almost uniform, and in most cases, the arithmetic coder will output a single bit for any single bit of input. Therefore the above alphabet encoding technique is more efficient than a technique which uses one bit to distinguish between case (1) and (2).

4.3 Representing the Sentinel

Since x may contain up to 256 different characters, we cannot represent the sentinel character \$ by a character of the ASCII alphabet. Instead, we implement it as an integer *sentinel* which points to a *virtual* character, that is larger than any character of the ASCII alphabet. The Burrows and Wheeler-Transformation of x is thus represented by a pair $(sentinel, \widetilde{x})$ where *sentinel* is the integer i such that $\varphi_x(i) = 1$, and $\widetilde{x}$ is defined as in Section 3, except that $\widetilde{x}_{sentinel}$ is undefined. For the suffix tree construction and the depth first traversal, we store x in an input buffer from index 1 to n and let $sentinel = n + 1$. Our implementation takes care that the virtual character, *sentinel* points to, is never compared to any character of the ASCII alphabet. Such a comparison is not necessary, since we always know it's result.

4.4 Implementation of the Suffix Tree

In [14] a very space efficient representation for suffix trees is described. It is based on linked lists, and requires $12n$ bytes in the worst case and about $10n$ bytes in practice. This is a considerable improvement over previous implementation techniques which require $28n$ bytes in the worst case and $20n$ bytes in practice, see [5, 11, 17, 18]. We have implemented McCreight's suffix tree construction [18] such that it produces the space efficient representation of [14] in $O(kn)$ time. To speed up the access to the successors and to facilitate a linear time depth first traversal, the linked list of the successors for each branching node $\overline{w}$ is ordered by $<_{\overline{w}}$. Additionally, for the *root* we store an $\mathcal{X}$-indexed table which allows to access the successors of the *root* in constant time. This table requires just k extra integers and considerably speeds up the suffix tree construction for large alphabets.

An alternative representation of the suffix tree uses a hash table to store the edges, as recommended in [18]. Unfortunately, this representation does not directly allow the depth first traversal to run in linear time. As already remarked in [15], an additional step is required to sort the edges lexicographically. This can be done by a bucket sorting algorithm, and thus requires linear time. We have implemented such an approach, but it proved to be considerably slower than directly computing the linked list representation. The construction of the hash table representation of ST was about as fast as the construction of the linked list representation of [14], but the additional sorting step was very slow.

4.5 Depth First Traversal

Implementing a depth first traversal of the suffix tree by a recursive procedure is straight forward. However, in the worst case the deepest branching node of the suffix tree can have $n-1$ predecessors on the path from the *root* (e.g. if $x = a^n$). This means that a recursive procedure would recurse to depth $n-1$, and the internal stack would require space for at least n extra integers. We cannot afford this space, and so we have implemented an iterative depth first traversal procedure. During the traversal, some parts of the suffix tree representation are not used any more. We have organized our procedure such that it reclaims these parts for its stack space. The iterative procedure is thus more space efficient and it proved to be faster than a recursive procedure. We also store x in the unused parts of the suffix tree representation. This allows to use the space for x to store the output $\widetilde{x}$. The suffix tree based method to construct $\widetilde{x}$ thus takes $O(kn)$ time and the only space it requires is the space for the suffix tree representation.

4.6 Move-to-Front Transformation

Without actually knowing the context tree modeling the tree source, the Burrows and Wheeler-Transformation permutes the input sequence in such a way that characters with the same right context are grouped together. Consider the jth context and let $\mathcal{X}_j$ denote the set of characters in x with this context. Because a context restricts the choice of the characters preceeding it, the size of the set $\mathcal{X}_j$ is usually small. Of course, $\mathcal{X}_j$ and $\mathcal{X}_{j+1}$ may be different. However, since the contexts are in lexicographic order, the difference between $\mathcal{X}_j$ and $\mathcal{X}_{j+1}$ is usually not too large, i.e. there is local stability. Unfortunately, we cannot immediately exploit this local stability, since we do not know when the contexts switch. For this reason, we transform the local stability into a global one using a move-to-front transformation. The idea of this transformation is to replace each symbol c by the number of distinct symbols which occurred since the last occurrence of c.

Let $a_1, \ldots, a_k$ be the characters in $\mathcal{X}$ in lexicographic order. For each $w \in \mathcal{X}^*$ and each permutation uav of $a_1 \ldots a_k$, with $u, v \in \mathcal{X}^*$ and $a \in \mathcal{X}$, we specify the function *mtf* by the following equations:

$$\begin{aligned} \mathit{mtf}(uav, \varepsilon) &= \varepsilon && (11) \\ \mathit{mtf}(uav, aw) &= |u| \cdot \mathit{mtf}(auv, w) && (12) \end{aligned}$$

The operator $\cdot$ denotes the concatenation of sequences. We define $\mathit{mtf}(x) = \mathit{mtf}(a_1 \ldots a_k, x)$ for any $x \in \mathcal{X}^*$. If $x \in \mathcal{X}^n$, then $\mathit{mtf}(x)$ is a sequence of length n over the alphabet $\mathcal{X}_{\mathrm{mtf}} = [0, k-1]$. $\mathit{mtf}(x)$ is the *move-to-front transformation* of x.

Example 8 Let $\mathcal{X} = \{a, b, c, d\}$ and $x = ccabbaaad$. Then $\mathit{mtf}(x)$ is computed

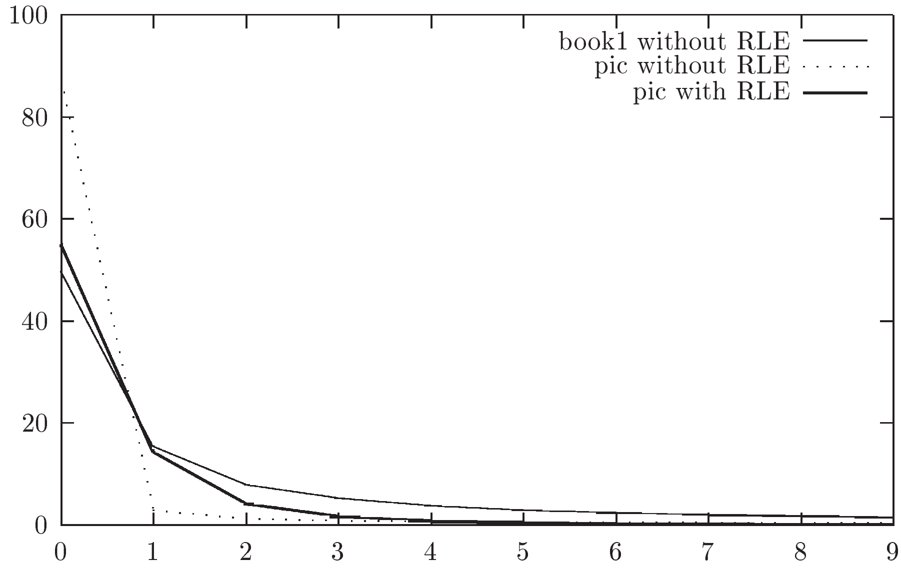

Figure 4: Relative frequencies of characters $[0, 9]$ in $\mathit{mtf}(\tilde{x})$

by the following steps, in which the ith application of Equation (12) is written as $ux_iv \xrightarrow{x_i,|u|} x_iuv$:

$$abcd \xrightarrow{c,2} cabd \xrightarrow{c,0} cabd \xrightarrow{a,1} acbd \xrightarrow{b,2} bacd \xrightarrow{b,0} bacd \xrightarrow{a,1} abcd \xrightarrow{a,0} abcd \xrightarrow{a,0} abcd \xrightarrow{d,3} dabc$$

Hence 201201003 is the move-to-front transformation of x.

One easily verifies that $\mathit{mtf}(x)$ can be computed in $O(kn)$ time. Moreover, given $\mathit{mtf}(x)$ one can compute x with the same complexity. Typically, $\mathit{occ}_{\mathit{mtf}(\tilde{x})}(a)$ monotonically decreases while a increases. This is e.g. shown in Figure 4 for the files *book1* and *pic* of the Calgary Corpus.

4.7 Zero Run Transformation

0 is the dominating symbol in $\mathit{mtf}(\tilde{x})$, and so there are many runs of the symbol 0 (0-*runs*, for short). Since it is better to not encode the 0's but the 0-runs, we apply a transformation to $\mathit{mtf}(\tilde{x})$, the 0-*run transformation*. Let $\mathcal{X}_0 = \{0_a, 0_b\}$ be an alphabet such that $(\mathcal{X}_{\mathrm{mtf}} \setminus \{0\}) \cap \mathcal{X}_0 = \emptyset$. We define a function $\zeta : \mathbb{N} \to \mathcal{X}_0^+$ by $\zeta(m) = w$ if and only if w is the mth sequence in the lexicographic order of all non-empty sequences over $\mathcal{X}_0$. Obviously, ζ is bijective. Table 4 shows $\zeta(m)$ for $m \in [1, 12]$. Let $y \in \mathcal{X}_{\mathrm{mtf}}^*$ and replace each maximal 0-run in y of length m, for some $m \in \mathbb{N}$, by the sequence $\zeta(m)$. Each symbol in y different from 0 remains unchanged. The resulting sequence, denoted by $\mathit{rle0}(y)$, is the 0-*run*

m	$\zeta(m)$	m	$\zeta(m)$	m	$\zeta(m)$	m	$\zeta(m)$
1	0_a	2	0_b	3	0_a0_a	4	0_a0_b
5	0_b0_a	6	0_b0_b	7	$0_a0_a0_a$	8	$0_a0_a0_b$
9	$0_a0_b0_a$	10	$0_a0_b0_b$	11	$0_b0_a0_a$	12	$0_b0_a0_b$

Table 4: $\zeta(m)$ for $m \in [1, 12]$

transformation of y, and it a sequence over the alphabet $\mathcal{X}_{rle0} = (\mathcal{X}_{\mathrm{mtf}} \setminus \{0\}) \cup \mathcal{X}_0$. It is easy to see that $rle0(y)$ can be computed in $O(n)$ time.

Note that a 0-run can have arbitrary length, so that encoding the length of a 0-run is the problem of universal coding of integers (see e.g. [2, 6, 16, 20, 24]). However, in our context the problem is simplified: (i) each 0-run in y is delimited by a symbol different from 0 and (ii) each encoding of a 0-run in $rle0(y)$ consists of the characters 0_a and 0_b, and it is delimited by a character different from 0_a and 0_b. Thus y can uniquely be decoded from $rle0(y)$ in linear time. Notice that in practice the global stability achieved by the move-to-front transformation is retained by the 0-run transformation.

4.8 A Hierarchical Model for Estimating and Forgetting

We have developed a simple hierarchical model for estimating probabilities in order to encode a sequence over the alphabet $\mathcal{X}_{rle0}$ by arithmetic coding. The idea is to partition $\mathcal{X}_{rle0}$ into disjoint classes. On the first level we estimate the probability $P_E^1(C)$ that the next character belongs to a certain class C. On the second level, we estimate, for a given class C, the probability $P_E^2(a \mid C)$ that $a \in C$ is the next character.

We define three singleton classes $C_c = \{c\}$ for each $c \in \{0_a, 0_b, 1\} \subseteq \mathcal{X}_{rle0}$. The remaining set $[2, k-1] \subseteq \mathcal{X}_{rle0}$ of characters is split into disjoint classes C_i of 2^{i-1} consecutive characters for $i = 2, 3, \ldots$: character 2 and 3 form class C_2, character 4-7 form class C_3, etc. If we have constructed class C_q and there are less than 2^q remaining characters in $\mathcal{X}_{rle0}$, then we add these to class C_q. Thus the last class C_q may consist of more than 2^{q-1} characters. Let $\mathcal{C} = \{C_{0_a}, C_{0_b}, C_1, C_2, \ldots, C_q\}$ be the collection of all classes as defined above. Let $\mathcal{C}(a)$ denote class $C \in \mathcal{C}$ if and only if $a \in C$.

Let $y = mtf(\tilde{x})$. When we process $rle0(y)$ from left to right, we do not know where the contexts change, or in other words, where we have to forget the characters previously processed. We tackle this problem by a technique which allows to forget parts of the previously processed sequence. In other words, we gradually change contexts. The idea is to accumulate each occurrence of a character by updating some statistics. For the first level, there is a statistic $S : \mathcal{C} \to \mathbb{N}$. For the second level, there are statistics $S_C : C \to \mathbb{N}$ for any $C \in \mathcal{C}$. All statistics are initialized to 1. For each processed character a, we increment $S(\mathcal{C}(a))$ by some constant l^1_{min}. If $a \geq 2$, then we additionally increment $S_{\mathcal{C}(a)}(a)$ by some constant l^2_{min}. If $S(\mathcal{C}(a))$ becomes larger than some constant l^1_{max}, then we set $S(C) := \lfloor (S(C)+1)/2 \rfloor$ for any $C \in \mathcal{C}$. If additionally $a \geq 2$ and $S_{\mathcal{C}(a)}(a)$ becomes larger than some constant l^2_{max}, then we set $S_{\mathcal{C}(a)}(b) := \left\lfloor (S_{\mathcal{C}(a)}(b)+1)/2 \right\rfloor$ for any $b \in \mathcal{C}(a)$. The choice of the constants l^1_{min} and l^2_{min} determines how fast the statistics grow. The larger l^1_{max} and l^2_{max}, the longer is the influence of some previously processed character. In practice we choose $l^1_{\text{min}} = 9$, $l^1_{\text{max}} = 243$, $l^2_{\text{min}} = 2$, and $l^2_{\text{max}} = 231$.

Consider the statistics after processing some prefix z of $rle0(y)$. Then we define our estimators P^1_E and P^2_E as follows:

$$\begin{aligned} P^1_E(C) &= \frac{S(C)}{\sum\limits_{C' \in \mathcal{C}} S(C')} \\ P^2_E(a \mid C) &= \frac{S_C(a)}{\sum\limits_{b \in C} S_C(b)} \end{aligned}$$

These probability estimates can be computed for the entire sequence $rle0(y)$ in $O(k)$ space and $O(kn)$ time. If l^1_{max} and l^2_{max} are large enough, so that the statistics are never halved, then we have $S(C) = 1 + l^1_{\text{min}} \cdot occ_z(C)$ and $S_C(b) = 1 + l^2_{\text{min}} \cdot occ_z(b)$ for any $C \in \mathcal{C}$ and any $b \in C$. Hence we obtain

$$P^1_E(C) = \frac{S(C)}{\sum\limits_{C' \in \mathcal{C}} S(C')} = \frac{1 + l^1_{\text{min}} \cdot occ_z(C)}{\sum\limits_{C' \in \mathcal{C}} (1 + l^1_{\text{min}} \cdot occ_z(C'))} = \frac{occ_z(C) + \frac{1}{l^1_{\text{min}}}}{occ_z(\mathcal{X}_{rle0}) + \frac{|\mathcal{C}|}{l^1_{\text{min}}}} \tag{13}$$

and with a similar equality chain we derive

$$P^2_E(a \mid C) = \frac{occ_z(a) + \frac{1}{l^2_{\text{min}}}}{occ_z(C) + \frac{|C|}{l^2_{\text{min}}}} \tag{14}$$

Thus for a binary alphabet and $l^i_{\text{min}} = 2$, we obtain the KT-estimator (see Section 2.4). In general, our estimator is the $\frac{1}{l^i_{\text{min}}}$ *biased k-array Dirichlet estimator* [22, 23]. For P^2_E, the probability of a symbol that has occurred once, is as likely as the sum of the probabilities of $1 + l^2_{\text{min}}$ symbols that have never occurred. For P^1_E the corresponding holds. Hence dividing l^i_{min} and l^i_{max} by their greatest common divisor would lead to a different estimator. This is already obvious from (13) and (14) in case l^i_{max} is large enough.

file	*length*	$\lvert\mathcal{X}\rvert$	LZ77	*gzip*	PPMC	BW94	*bzip2*	BK98
bib	111261	81	3.75	2.55	2.11	2.07	1.97	1.93
book1	768771	82	4.57	3.28	2.48	2.49	2.42	2.31
book2	610856	96	3.93	2.74	2.26	2.13	2.06	2.00
geo	102400	256	6.34	5.34	4.78	4.45	4.44	4.49
news	377109	98	4.37	3.10	2.65	2.59	2.51	2.48
obj1	21504	256	5.41	3.84	3.76	3.98	4.01	3.86
obj2	246814	256	3.81	2.63	2.69	2.64	2.47	2.45
paper1	53161	95	3.94	2.82	2.48	2.55	2.49	2.45
paper2	82199	91	4.10	2.92	2.45	2.51	2.43	2.38
pic	513216	159	2.22	0.82	1.09	0.83	0.77	0.73
progc	39611	92	3.84	2.70	2.49	2.58	2.53	2.50
progl	71646	87	2.90	1.81	1.90	1.80	1.74	1.70
progp	49379	89	2.93	1.83	1.84	1.79	1.73	1.69
trans	93695	99	2.98	1.63	1.77	1.57	1.52	1.47
	3251493		3.94	2.72	2.48	2.43	2.36	2.32

Table 5: Compression rates in bits/byte for different programs

5 Experimental Results

We implemented a data compression program in C, which employs the previously described methods and implementation techniques. In a first experiment we measured the compression rate in bits/byte for 14 files of the Calgary Corpus, and compared it to other programs. Table 5 shows the results for the following programs: LZ77 (see [29]), *gzip* with option –9 (see [8]), PPMC (see [19]), BW94 (see [3]), *bzip2* with option –9 (see [21]), and our program which is referred to by BK98. LZ77 and *gzip* are sequential data compression programs based on the Lempel-Ziv Algorithm, PPMC is based on statistical modeling, and the remaining programs use the Burrows and Wheeler-Transformation. The last row of Table 5 shows the sum of the lengths of the files and for each program the average compression rate over all 14 files. In each row the best compression rate is shown in a grey box. Except for three files, our program achieves the best compression rate. It also achieves the best average compression rate.

To demonstrate the practical relevance of our program, we performed some time and space measurements for the same 14 files of the Calgary Corpus. It is interesting to note that the computation of the Burrows and Wheeler-Transformation accounts, on average, for about 80% of the running time of our program, and it dominates the space requirement (see Section 4.5). Table 6 shows compression time (*ctime*) and decompression time (*dtime*) for *gzip* and for our program. It

		gzip		BK98		
file	*length*	*ctime*	*dtime*	*ctime*	*dtime*	*cspace*
bib	111261	0.40	0.05	0.86	0.32	1.16
book1	768771	3.77	0.28	8.17	2.63	8.32
book2	610856	2.14	0.20	5.69	1.87	6.52
geo	102400	1.00	0.07	2.29	0.63	0.87
news	377109	1.03	0.13	4.36	1.28	3.97
obj1	21504	0.07	0.03	0.24	0.12	0.19
obj2	246814	1.17	0.09	2.89	0.82	2.53
paper1	53161	0.15	0.04	0.43	0.18	0.57
paper2	82199	0.32	0.04	0.68	0.28	0.89
pic	513216	3.00	0.10	1.52	0.45	5.04
progc	39611	0.12	0.03	0.32	0.15	0.42
progl	71646	0.26	0.04	0.49	0.20	0.80
progp	49379	0.20	0.04	0.32	0.14	0.56
trans	93695	0.21	0.04	0.61	0.24	1.07
	3251493	13.84	1.18	28.87	9.31	32.91

Table 6: Running times (in seconds) and space requirement (in megabytes)

also shows the space our program requires for compression (*cspace*). *gzip*'s space requirement it limited to 64 kilobytes, so we do not show it here. The last row of Table 6 gives the sums of the corresponding columns. The results were obtained on a computer with Pentium processor (166 MHz, 64 MB RAM) under the operating system Linux. We used the *gcc* compiler, version 2.7.2.2 with the optimizing option –O3. Times are user times in seconds (averaged over ten runs) as reported by the *gnu time* utility. Space is in megabytes. The table shows that *gzip* compresses about twice as fast as our program does: *gzip* compresses at a rate of 234934 characters per second, while our program achieves a rate of 112625 characters per second. *gzip* decompresses 7.9 times faster than our program does. The average space consumption of our program is about $10n$ bytes.

In a final experiment we applied *gzip* and our program to DNA data. This domain becomes more and more important for data compression programs, since the size of the sequence data bases grow exponentially. We randomly selected a continuous DNA sequence of the yeast genome (length 10^6, alphabet `A,C,G,T`). For this DNA sequence *gzip* achieves a compression rate of 2.24 bits/byte with a running time of 33.91 seconds, while our program achieves a compression rate of 2.04 bits/byte using 8.65 seconds. This demonstrates that our program sometimes clearly outperforms *gzip* in compression rate as well as compression time.

Acknowledgements

We wish to thank J. Åberg, Y.M. Shtarkov, T.J. Tjalkens, and F.M.J. Willems for fruitful discussions on our work.

References

[1] N. Abramson. *Information Theory and Coding*, pages 61–62. McGraw-Hill, New York, NY, 1963.

[2] R. Ahlswede, T.S. Han, and K. Kobayashi. Universal Coding of Integers and Unbounded Search Trees. *IEEE Transactions on Information Theory*, **43**(2):669–682, 1997.

[3] M. Burrows and D. J. Wheeler. A Block-Sorting Lossless Data Compression Algorithm. Research Report 124, Digital Systems Research Center, 1994. `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html`.

[4] T. Cover and J. Thomas. *Elements of Information Theory.* John Wiley & Sons, New York, NY, 1991.

[5] M. Crochmore and R. Vérin. Direct Construction of Compact Acyclic Word Graphs. In *Proc. of the Annual Symposium on Combinatorial Pattern Matching* (CPM'97), LNCS **1264**, pages 116–129, 1997.

[6] P. Elias. Universal Codword Sets and Representation of Integers. *IEEE Transactions on Information Theory*, **21**:194–203, 1975.

[7] M. Farach. Optimal Suffix Tree Construction with Large Alphabets. In *Proceedings of the 38th Annual Symposium on the Foundations of Computer Science, FOCS 97*, New York, 1997. IEEE Comput. Soc. Press. `ftp://cs.rutgers.edu/pub/farach/Suffix.ps.Z`.

[8] J.L Gailly. The gzip program, version 1.2.4,, 1993. `ftp://prep.ai.mit.edu/pub/gnu/gzip-1.2.4.tar.gz`.

[9] R. Giegerich and S. Kurtz. A Comparison of Imperative and Purely Functional Suffix Tree Constructions. *Science of Computer Programming*, **25**(2-3):187–218, 1995.

[10] R. Giegerich and S. Kurtz. From Ukkonen to McCreight and Weiner: A Unifying View of Linear-Time Suffix Tree Construction. *Algorithmica*, **19**:331–353, 1997.

[11] R.W. Irving. Suffix Binary Search Trees. Research Report, Department of Computer Science, University of Glasgow, 1996. `http://www.dcs.gla.ac.uk/~rwi/papers/sbst.ps`.

[12] F. Jelinek. *Probabilistic Information Theory*, pages 476–489. McGraw-Hill, New York, NY, 1968.

[13] R.E. Krichevsky and V.K. Trofimov. The Performance of Universal Encoding. *IEEE Transactions on Information Theory*, **27**:199–207, 1981.

[14] S. Kurtz. Reducing the Space Requirement of Suffix Trees. Manuscript, 1998.

[15] N.J. Larson. The Context Trees of Block Sorting Compression. In *Proceedings of the IEEE Data Compression Conference*, 1998.

[16] V.I. Levenshtein. On the Redundancy and Delay of Decodable Coding of Natural Numbers. *Probl. Cybern.*, **20**:173–179, 1968. (in Russian, translated by Y.M. Shtarkov).

[17] U. Manber and E.W. Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, **22**(5):935–948, 1993.

[18] E.M. McCreight. A Space-Economical Suffix Tree Construction Algorithm. *Journal of the ACM*, **23**(2):262–272, 1976.

[19] A. Moffat. Implementing the PPM Data Compression Scheme. *IEEE Trans. on Communications*, **28**(11):1917–1921, 1990.

[20] J.J. Rissanen. A Universal Prior for Integers and Estimation by Minimum Description Length. *Ann. Statist.*, **11**:416–431, 1983.

[21] J. Seward. The bzip2 program, version 0.1pl2, 1997. `http://www.muraroa.demon.co.uk`.

[22] Y.M. Shtarkov. Universal Sequential Coding of Single Messages. *Problems Inform. Transm.*, **23**(3):3–17, 1987.

[23] Y.M. Shtarkov, T.J. Tjalkens, and F.M.J. Willems. Multialphabet Coding of Memoryless Sources. *Problems Inform. Transm.*, **31**(2):20–35, 1995.

[24] Q.F. Stout. Improved Prefix Encodings of Natural Numbers. *IEEE Transactions on Information Theory*, **26**:607–609, 1980.

[25] E. Ukkonen. On-line Construction of Suffix-Trees. *Algorithmica*, **14**(3), 1995.

[26] P. Weiner. Linear Pattern Matching Algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, The Univsersity of Iowa, 15-17 October, 1973.

[27] F.M.J. Willems, Y.M. Shtarkov, and T.J. Tjalkens. The Context-Tree Weighting Method: Basic Properties. *IEEE Transactions on Information Theory*, **41**:653–664, 1995.

[28] I.H. Witten, R.M. Neal, and J.G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM*, **30**(6):520–540, 1987.

[29] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, **23**(3):337–343, 1977.