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

Bernhard Balkenhol and Stefan Kurtz

Abstract—A very interesting recent development in data compression is the Burrows-Wheeler Transformation [1]. 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-Wheeler Transformation from an information theoretic point of view. Based on this analysis, the main part of the paper systematically considers techniques to efficiently implement a practical 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.

Index Terms—Lossless data compression, Burrows-Wheeler Transformation, context trees, suffix trees.

1 Introduction and Overview

A very interesting recent development in data compression is the Burrows-Wheeler Transformation [1]. 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-Wheeler Transformation is fast and it leads to good compression results.

So far the Burrows-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 [2], we will show that the Burrows-Wheeler Transformation permutes the output sequences of the source in such a way that the permutation can be partitioned into intervals, one for each leaf of the context tree. Due to this property of the context tree, the subsequence of the source in each such interval is i.i.d. As a consequence, for known context trees, a data compression scheme based on the Burrows-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-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 when the 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-Wheeler Transformation in linear time and space using suffix trees.
- We provide a technique to efficiently encode runs of zeros after the move-to-front transformation.
- We describe a 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 also 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 [3], [4].

We have developed a data compression program that employs these implementation techniques. It runs in $O(kn)$ time and requires $O(n)$ space, where n is the length of the input sequence and k is the alphabet size. Experimental results show that it achieves a better compression rate than other programs for most files of the Calgary Corpus [5] and the Canterbury Corpus [6]. It also showed the best average compression rate (2.32 bits/byte for the Calgary Corpus and 2.05 bits/byte for the Canterbury Corpus). For the former corpus, the *gzip*-program [7] compresses 2.4 times faster

- *B. Balkenhol is with the Fakultät für Mathematik, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany. E-mail: bernhard@mathematik.uni-bielefeld.de.*
- *S. Kurtz is with the Technische Fakultät, Universität Bielefeld, Postfach 100 131, 33501 Bielefeld, Germany. E-mail: kurtz@techfak.uni-bielefeld.de.*

Manuscript received 30 Sept. 1998; accepted 26 May 2000.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 107476.

than our program. For the latter corpus, our program was 1.5 times faster than *gzip*.

This paper is organized as follows: In Section 2, we carefully establish basic notions and review some basic properties of context trees. Section 3 is devoted to the Burrows-Wheeler Transformation. We carefully define the transformation and state its properties in Section 3.1. Sections 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.

This paper extracts the core of a much wider report [8], where we give more details on the information theoretic background of our work (i.e., arithmetic coding [9], coding redundancy, and Krichevsky-Trofimov estimated probabilities [10], [3]), and present more examples to ease understanding of our techniques.

2 Preliminaries

For any numbers $l, r \in \mathbb{N}_0$, $[l,r]$ denotes the set $\{i \in \mathbb{N}_0 : l \leq i \leq r\}$. 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 $s = s_1 \dots s_n$, where $s_i \in \mathcal{S}$. $s_n \dots s_1$, denoted by s^{-1}, is the *reverse* of $s = s_1 \dots s_n$. If $i \leq j$, then $s_i \dots s_j$ is the *factor* of s beginning with the ith element and ending with the jth element. If $i > j$, then $s_i \dots s_j$ 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 total order on $\mathcal{X}$ is denoted by $\prec$. The elements of $\mathcal{X}$ are *characters* or *symbols*. If convenient, we denote the characters by their *ranks* 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 $S \subseteq \mathcal{X}$.

An $\mathcal{X}^+$-*tree* T is a finite rooted tree with edge labels from $\mathcal{X}^+$. The empty $\mathcal{X}^+$-tree consists only of the *root*. For each $a \in \mathcal{X}$, every node v in T has at most one outgoing 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 *internal 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. An $\mathcal{X}^+$-tree is *atomic* if every edge is labeled by a *single* character from $\mathcal{X}$. An $\mathcal{X}^+$-tree is *compact* if every node is the *root*, a leaf, or a branching node. An atomic as well as a compact $\mathcal{X}^+$-tree T is uniquely determined by $words(T)$.

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.

A *context tree* CT is an atomic $\mathcal{X}^+$-tree such that each internal node has exactly k outgoing edges. Each leaf $\overline{c}$ is labeled by a probability distribution $P_{CT}(\cdot|c^{-1})$. 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

$$P_A(x) = P_A(x_1 \dots x_{l(x)}) \prod_{i=l(x)+1}^{n} P_{CT}(x_i \mid (c_i)^{-1}), \tag{2}$$

where 1) $l(x)$ is the smallest integer $i \in [1,n]$ such that $(x_1^i)^{-1}$ is a leaf in CT and 2) 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 (2) for any $x \in \mathcal{X}^n$ is called the *model of the tree source*.

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, i.e., the source is stationary. 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 leaves 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 using some locally adaptive statistical compression scheme, like arithmetic coding.

3 The Burrows-Wheeler Transformation

In this section, we introduce the Burrows-Wheeler Transformation and study its properties. We explain why we define it differently from the original transformation in [1]. We show how to construct the transformation and how to decode it in linear time and space. The idea of the Burrows-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 largest character in $\mathcal{X}$.

For any $i \in [1, n+1]$, let $S_x(i) = x_i \dots x_n\$$ denote the ith nonempty 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 nonempty 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 largest 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-Wheeler Transformation* of x is the sequence $\tilde{x}$ of length $n+1$ such that, for any $i \in [1, n+1]$:

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

Note that Burrows and Wheeler [1] define their transformation in a slightly different way. They consider all cyclic shifts $x_i x_{i+1} \dots x_n x_1 \dots x_{i-1}$ of x and sort them lexicographically. If one writes the cyclic shifts line by line, beginning with the smallest one, then the last column of the resulting matrix is the original Burrows-Wheeler Transformation. Burrows and Wheeler later also appended a sentinel character to x (as we do), recognizing the fact that it is more efficient to sort nonempty suffixes than cyclic shifts since one can stop the pairwise character comparisons as soon as one sees the sentinel to the right of x_n. Another reason for appending the sentinel is that it prevents us 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-Wheeler Transformation.

Example 1. Let $\mathcal{X} = \{a, b\}$ and $x = abab$. The following table shows the nonempty suffixes of $x\$$ in lexicographic order and the Burrows-Wheeler Transformation of x:

	ordered suffixes					$\tilde{x}$
$S_x(1)$	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, $\tilde{x} = \$baab$. To obtain the Burrows-Wheeler Transformation according to the definition in [1], 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-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]

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

3.1 Properties

In this section, we show that the Burrows-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. Similar observations were previously made by other authors (e.g., [11]), but not stated and proven formally.

Theorem 1. *Suppose that the source is a tree source with a model CT. Let $r = |\mathcal{L}(CT)|$ and $c_1, \dots, c_r$ be the leaves 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 $w_1, \dots, w_r$ such that:*

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

Proof. The first $l(x)$ characters in x do not have a context w.r.t. CT, see (2). For this reason we delete them from $\widetilde{y}$. In contrast, 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 inCT such that c is a prefix of the suffix $y_{n+2-i} y_{n+2-i+1} \dots y_n$ of y. For this reason, we 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 nonempty 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 of the context tree as a prefix. This also partitions z into factors $w_1, \dots, w_r$ such that $z = w_1 \dots w_r$. 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,

$$\begin{aligned} w_j &= y_{(n+2-i_1)-1} y_{(n+2-i_2)-1} \cdots y_{(n+2-i_{l_j})-1} \\ &= y_{n+1-i_1} y_{n+1-i_2} \cdots y_{n+1-i_{l_j}} = x_{i_1} x_{i_2} \dots x_{i_{l_j}}. \end{aligned}$$

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. □

Example 2. Let $x = 100100110$ and consider a context tree with the leaves 00, 01, and 1. Then,

$$\begin{gathered} n = 9, \\ l(x) = 1, \\ Sub(x, 00) = \{7, 4\}, \\ Sub(x, 01) = \{6, 3\}, \end{gathered}$$

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 1) into factors $w_1, \ldots, w_r$ and encode each w_j as described at the end of Section 2. In this way, it is possible to achieve the same compression rate as a method which does without the Burrows-Wheeler Transformation. However, such a method requires storing, at each leaf of CT, a statistic. In contrast, after the Burrows-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-Wheeler Transformation is that it allows us 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 of 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-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-Wheeler Transformation $\tilde{x}$, one first computes the suffix order on $x\$$. In [1], 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 [12], [13], [14], [15]. 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 other methods [16], [17] whose worst case running time is $O(n \log n)$. We refer to these as *nonlinear methods*.
- In [18], it was recently shown that the suffix tree for x can be computed in $O(kn)$ time using about $10n$ bytes of space in the average case. The space consumption of a suffix tree based method is thus comparable to the nonlinear methods which require $8n$ bytes [16] and $9n$ bytes [17].
- 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 $\tilde{x}$.

The *suffix tree* for x, denoted by ST, is the compact $\mathcal{X}^+$-tree T such that

$$words(T) = \{w \in \mathcal{X}^* \mid w \text{ is a factor of } x\$\}.$$

Due to the sentinel character, there is a one-to-one correspondence between the leaves of ST and the nonempty suffixes of $x\$$: Each suffix $S_x(i)$ is represented by the leaf $\overline{S_x(i)}$ and different leaves 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 internal node in ST is branching. This means that there are at most n internal 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 leaves of ST and the nonempty suffixes of $x\$$, the Burrows-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 $\prec_{\overline{w}}$, which is defined as follows:

$$\overline{w} \xrightarrow{au} \overline{wau} \prec_{\overline{w}} \overline{w} \xrightarrow{cv} \overline{wcv} \iff a \prec c.$$

That is, the edges are sorted according to the first character of each edge label. Since no two edges outgoing from $\overline{w}$ have a label beginning with the same character, $\prec_{\overline{w}}$ is a total order on the set of all edges outgoing from $\overline{w}$. 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 $S_x(i) \prec S_x(j)$, where $\prec$ is the lexicographic order on $\mathcal{X}^*$. 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 leaves 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 $\prec_{\overline{w}}$, then the depth first traversal runs in linear time. No extra space is needed, except for the output sequence $\tilde{x}$.

Linear time suffix constructions have a long history, starting with the construction of Weiner [12]. Later authors [13], [14] have developed improved algorithms. Giegerich and Kurtz [19] reveal that these three linear time algorithms are very closely related, although they are all based on rather different intuitive ideas. Recently, Farach [15] described a linear time algorithm which differs very much from the other algorithms.

2. Burrows and Wheeler [1] report that they have implemented a suffix tree-based method to compute the Burrows-Wheeler Transformation. However, they do not give enough information to substantially evaluate how their implementation performs in comparison to the nonlinear methods.

For our particular application, we consider McCreight's algorithm [13] 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 [20]. We have not seen any practical results of the space and time behavior of Farach's algorithm. 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-Wheeler Transformation.

3.3 Decoding

Since our definition of the Burrows-Wheeler Transformation slightly differs from the original, we now present an algorithm to decode x given $\tilde{x}$. The algorithm runs in $O(n)$ time and space and is divided into three phases. It is similar to the algorithm given in [1].

In the first phase of the decoding algorithm, two tables $count : \mathcal{X} \to [0, n]$ and $offset : [1, n+1] \to [1, n]$ 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 $r \in [1, n+1]$, $offset[r]$ is the number of positions $p \in [1, r]$ such that $\tilde{x}_p = \tilde{x}_r$. That is, $offset[r] = l$ if and only if position r is the lth position in $\tilde{x}$ (from left to right) where character $\tilde{x}_r$ occurs.

Note that $\tilde{x}$ is just a permutation of $x\$$. Hence, $count$ can be computed in one pass over $\tilde{x}$. 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 $a \in \mathcal{X}$,

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

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

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_ith smallest nonempty 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 $\tilde{x}$ due to the following property:

$$x_i = x_{i+1-1} = x_{\varphi_x(r_{i+1})-1} = \tilde{x}_{r_{i+1}}. \quad (3)$$

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

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

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

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 $a = \tilde{x}_{r_{i+1}} \neq \$$. One easily observes that $base[a] + 1 \leq r_i \leq base[a] + count[a]$. If $S_x(i)$ is the only suffix beginning with a, then $count[a] = offset[r_{i+1}] = 1$. Hence, $r_i = base[a] + 1 = base[a] + offset[r_{i+1}]$. Now, suppose there is a suffix $S_x(i')$, $i' \in [1, n]$, $i' \neq i$, which also begins with a. Then, $a = x_{i'} = \tilde{x}_{r_{i'+1}}$. Moreover, we have

$$\begin{aligned} S_x(i) \prec S_x(i') &\iff S_x(i+1) \prec S_x(i'+1) \\ &\iff offset[r_{i+1}] < offset[r_{i'+1}]. \end{aligned}$$

Hence, if $offset[r_{i+1}] = l$, then $S_x(i)$ is the lth suffix beginning with a. This implies $r_i = base[a] + offset[r_{i+1}]$ □

With Property (3) and Lemma 1, it is easy to show that the following algorithm correctly decodes x from $\tilde{x}$ in $O(n)$ time and space.

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

The algorithm needs $n+1$ integers for table $offset$, $2k$ integers for tables $count$ and $base$, and $2n+1$ characters to store the input $\tilde{x}$ 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 and engineering aspects. We describe techniques to efficiently implement a data compression scheme based on the Burrows-Wheeler Transformation. We will always motivate why we chose the particular technique. If necessary, we modify it and show how to make it work well in practice. The structure of this section follows the data flow of our data compression program, as depicted in Fig. 1. For lack of space, we do not describe the last phase, i.e., arithmetic coding. The interested reader is referred to [9].

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

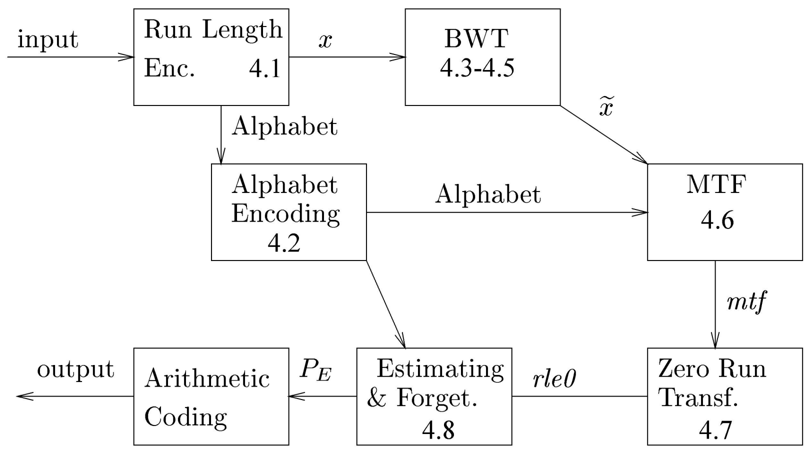

Fig. 1. The data flow in our compression program. The numbers refer to the sections in which the different phases are explained.

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-Wheeler Transformation in Section 3.1, we reversed the input sequence and then applied the transformation. In this way, we are consistent with other methods and we are able to conveniently describe its properties. However, to save computation time, our compression program directly computes the Burrows-Wheeler Transformation for the input sequence, possibly after applying the run length encoding.

4.1 Run Length Encoding

A *run* in x is a nonempty 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 percent zeros. If we instead first apply the run length encoding to x, then this reduces the number of zeros to about 60 percent. In the latter case, we can achieve much better estimates for the nonzero 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 only if 1) there is an ASCII character available which does not appear in x (this is used for marking the start of a run), and 2) if $|rle(x)| < 0.7n$ holds. In order to decide 1) and 2), 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 1) we have more free parameters for our estimator (which increases coding redundancy) and 2) 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 β which searches 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 $i \in [l, r]$, β is specified as follows:

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

The operator $\cdot$ denotes the concatenation of sequences. 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 the largest integer 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 $\beta(2(a'_l - b) + 2, 1, 2(256 - b) + 1)$ and $\beta(j - l, 1, 256 - a'_l)$. 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 $\beta(2(a'_l - b) + 1, 1, 2(256 - b) + 1)$. We proceed with $b = a'_l + 2$ and $l = l + 1$.

Note that, while b grows, the interval used for β becomes smaller. The alphabet is encoded before x. 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 Case 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-Wheeler Transformation of x is thus represented by a pair $(sentinel, \tilde{x})$, where $sentinel$ is the integer i such that $\varphi_x(i) = 1$ and $\tilde{x}$ is defined as in Section 3, except that $\tilde{x}_{sentinel}$ 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 its result.

4.4 Implementation of the Suffix Tree

In [18], a very space efficient representation for suffix trees is described. It is based on linked lists and requires about $10n$ bytes in practice. This is a considerable improvement over previous implementation techniques which require about $20n$ bytes in practice, see [13], [16], [21], [22]. We have implemented McCreight's suffix tree construction [13] such that it produces the space efficient representation of [18] 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 $\prec_{\overline{w}}$. Additionally, for the *root* we store an $\mathcal{X}$-indexed table which allows us 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 [13]. Unfortunately, this representation does not directly allow the depth first traversal to run in linear time. As already remarked in [23], 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 [18], 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 straightforward. 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 us to use the space for x to store the output $\tilde{x}$. The suffix tree based method to construct $\tilde{x}$ 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-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 preceding 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, see [24]. 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 $a_1, \dots, a_k$ be the characters in $\mathcal{X}$ in lexicographic order. For each $w \in \mathcal{X}^*$ and each permutation uav of $a_1 \dots a_k$, with $u, v \in \mathcal{X}^*$ and $a \in \mathcal{X}$, we specify the function mtf by the following equations:

$$mtf(uav, \varepsilon) = \varepsilon \tag{4}$$

$$mtf(uav, aw) = |u| \cdot mtf(auv, w). \tag{5}$$

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

Example 3. Let $\mathcal{X} = \{a, b, c, d\}$ and $x = ccabbaaad$. Then, $mtf(x)$ is computed by the following steps, in which the ith application of (5) is written as $ux_iv \xrightarrow{x_i, |u|} x_iuv$:

$$\begin{array}{llllll} abcd & \xrightarrow{c,2} & cabd & \xrightarrow{c,0} & cabd & \\ & \xrightarrow{a,1} & acbd & \xrightarrow{b,2} & bacd & \\ & \xrightarrow{b,0} & bacd & \xrightarrow{a,1} & abcd & \\ & \xrightarrow{a,0} & abcd & \xrightarrow{a,0} & abcd & \xrightarrow{d,3} \quad dabc \end{array}$$

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

One easily verifies that $mtf(x)$ can be computed in $O(kn)$ time. Moreover, given $mtf(x)$, one can compute x with the same complexity. Typically, $occ_{mtf(\widetilde{x})}(a)$ monotonically decreases while a increases. This is because the Burrows-Wheeler Transformation typically produces runs of any symbol, which become runs of zeros after the move-to-front transformation.

4.7 Zero Run Transformation

0 is the dominating symbol in $mtf(\widetilde{x})$ and, so, there are many runs of the symbol 0 (0-*runs*, for short). Since it is better to not encode the 0s, but the 0-runs, we apply a transformation to $mtf(\widetilde{x})$, the 0-*run transformation*. Let $\mathcal{X}_0 = \{0_a, 0_b\}$ be an alphabet such that $\mathcal{X}_{\text{mtf}} \cap \mathcal{X}_0 = \emptyset$. We define a function $\zeta : \mathbb{N} \rightarrow \mathcal{X}_0^+$ by $\zeta(m) = w$ if and only if w is the mth sequence in the lexicographic order of all nonempty sequences over $\mathcal{X}_0$. Obviously, ζ is bijective. Let $y \in \mathcal{X}_{\text{mtf}}^*$ 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 $rle0(y)$, is the 0-*run transformation* of y and it is a sequence over the alphabet $\mathcal{X}_{rle0} = (\mathcal{X}_{\text{mtf}} \setminus \{0\}) \cup \mathcal{X}_0$. 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., [25], [26], [27], [28], [29]). However, in our context, the problem is simplified: 1) Each 0-run in y is delimited by symbols different from 0 and 2) each encoding of a 0-run in $rle0(y)$ consists of the characters 0_a and 0_b and it is delimited by characters 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 (similar to [30]) 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 $i = 2, 3, \ldots$: Characters 2 and 3 form class C_2, characters 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 $y = mtf(\widetilde{x})$. 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 forgetting 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 $S : \mathcal{C} \rightarrow \mathbb{N}$. For the second level, there are statistics $S_C : C \rightarrow \mathbb{N}$ 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 $l_{\min}^1$. If $a \geq 2$, then we additionally increment $S_{\mathcal{C}(a)}(a)$ by some constant $l_{\min}^2$. If $S(\mathcal{C}(a))$ becomes larger than some constant $l_{\max}^1$, 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 $l_{\max}^2$, then we set $S_{\mathcal{C}(a)}(b) := \lfloor (S_{\mathcal{C}(a)}(b)+1)/2 \rfloor$ for any $b \in \mathcal{C}(a)$. The choice of the constants $l_{\min}^1$ and $l_{\min}^2$ determines how fast the statistics grow. The larger $l_{\max}^1$ and $l_{\max}^2$, the longer is the influence of some previously processed character. In practice, we choose $l_{\min}^1 = 9$, $l_{\max}^1 = 243$, $l_{\min}^2 = 2$, and $l_{\max}^2 = 231$.

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

$$P_E^1(C) = \frac{S(C)}{\sum_{C' \in \mathcal{C}} S(C')}$$

$$P_E^2(a \mid C) = \frac{S_C(a)}{\sum_{b \in C} S_C(b)}.$$

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

$$P_E^1(C) = \frac{occ_z(C) + \frac{1}{l_{\min}^1}}{occ_z(\mathcal{X}_{rle0}) + \frac{|\mathcal{C}|}{l_{\min}^1}} \tag{6}$$

$$P_E^2(a \mid C) = \frac{occ_z(a) + \frac{1}{l_{\min}^2}}{occ_z(C) + \frac{|C|}{l_{\min}^2}}. \tag{7}$$

Thus, for a binary alphabet and $l_{\min}^i = 2$, we obtain the Krichevsky-Trofimov estimator [3], [10]. In general, our estimator is the $(1/l_{\min}^i)$-*biased k-array Dirichlet estimator* [3], [4]. For P_E^2, the probability of a symbol that has occurred once is as likely as the sum of the probabilities of $1 + l_{\min}^2$ symbols that have never occurred. For P_E^1, the corresponding holds. Hence, dividing $l_{\min}^i$ and $l_{\max}^i$ by their greatest common divisor would lead to a different estimator. This is already obvious from (6) and (7) in case $l_{\max}^i$ is large enough.

TABLE 1
Compression Rates in Bits/Byte for the Calgary Corpus

file	*length*	*k*	*pack*	*compress*	*gzip*	DMC	PPM	*bred*	*bzip2*	*szip*	BK98
bib	111261	81	5.24	3.35	2.51	2.20	2.12	2.19	1.97	1.98	1.94
book1	768771	81	4.56	3.46	3.25	2.51	2.54	2.98	2.42	2.36	2.31
book2	610856	96	4.83	3.28	2.70	2.19	2.25	2.51	2.06	2.03	2.00
geo	102400	256	5.69	6.08	5.34	4.80	4.91	4.89	4.45	4.29	4.49
news	377109	98	5.23	3.86	3.06	2.77	2.68	2.94	2.52	2.50	2.49
obj1	21504	256	6.08	5.23	3.84	4.12	3.72	3.91	4.01	3.78	3.87
obj2	246814	256	6.30	4.17	2.63	2.76	2.52	2.67	2.48	2.48	2.46
paper1	53161	95	5.03	3.77	2.79	2.73	2.48	2.58	2.49	2.50	2.45
paper2	82199	91	4.65	3.52	2.89	2.59	2.45	2.58	2.44	2.44	2.38
pic	513216	159	1.66	0.97	0.82	0.82	0.99	0.82	0.78	0.82	0.74
progc	39611	92	5.26	3.87	2.68	2.75	2.48	2.58	2.53	2.52	2.50
progl	71646	87	4.81	3.03	1.80	1.99	1.84	1.79	1.74	1.75	1.71
progp	49379	89	4.91	3.11	1.81	2.00	1.80	1.78	1.74	1.82	1.70
trans	93695	99	5.58	3.27	1.61	1.92	1.72	1.56	1.53	1.59	1.48
	3141622		4.98	3.64	2.69	2.58	2.46	2.55	2.36	2.34	2.32

TABLE 2
Compression Rates in Bits/Byte for the Canterbury Corpus

file	*length*	*k*	*pack*	*compress*	*gzip*	DMC	PPM	*bred*	*bzip2*	*szip*	BK98
alice29	152089	74	4.62	3.27	2.85	2.38	2.31	2.55	2.27	2.25	2.23
ptt5	513216	159	1.66	0.97	0.82	0.82	0.99	0.82	0.78	0.82	0.74
fields	11150	90	5.12	3.56	2.24	2.40	2.11	2.17	2.18	2.19	2.11
kennedy	1029744	256	3.60	2.41	1.63	1.44	1.08	1.21	1.01	0.84	0.90
sum	38240	255	5.42	4.21	2.67	3.03	2.68	2.77	2.70	2.70	2.62
lcet10	426754	84	4.70	3.06	2.71	2.13	2.19	2.47	2.02	2.00	1.97
plrabn12	481861	81	4.58	3.38	3.23	2.48	2.48	2.89	2.42	2.38	2.36
cp	24603	86	5.30	3.68	2.59	2.69	2.38	2.50	2.48	2.44	2.43
grammar	3721	76	4.87	3.90	2.65	2.84	2.43	2.69	2.79	2.60	2.55
xargs	4227	74	5.10	4.43	3.31	3.51	3.00	3.26	3.33	3.25	3.11
asyoulik	125179	68	4.85	3.51	3.12	2.64	2.53	2.84	2.53	2.51	2.49
e.coli	4638690	4	2.25	2.17	2.24	2.10	2.03	2.16	2.16	2.07	2.04
bible	4047392	63	4.39	2.77	2.33	1.82	1.66	2.09	1.67	1.62	1.63
world192	2473400	94	5.04	3.19	2.33	1.83	1.66	2.24	1.58	1.60	1.56
	13970266		4.39	3.17	2.48	2.29	2.10	2.33	2.13	2.09	2.05

5 Experimental Results

We implemented a data compression program in C. It employs the previously described methods and implementation techniques. In a first experiment, we measured its compression rate in bits/byte for files of the Calgary Corpus [5] and the Canterbury Corpus[3] [6] and compared the rates to other programs, which have similar requirements in space and time. Tables 1 and 2 show the results for the programs *pack*, *compress*, *gzip* with option –9 (see [7]), DMC with memory usage of 16MB (see [31]), PPM with option *-o3* and escape method D (see [32]), *bred* (see [1]), *bzip2* with option –9 (see [33]), *szip* with block size 1.7MB (see [34]), and, finally, our program which is referred to by BK98. *pack* is the Unix-program using Huffman coding on a byte-by-byte basis. *compress* and *gzip* are sequential data compression programs based on [35] and [36], respectively. DMC is based on Dynamic Markov Compression. PPM is based on statistical modeling and the remaining programs use the Burrows-Wheeler Transformation. The last row of both tables shows the total length of the files and for each program the average compression rate. In each row, the best compression rate is shown in a gray box. For most files, our program achieves the best compression rate. Exceptions are mainly small files. For both corpora, our program shows the best average compression rate: 2.32 bits/byte for the Calgary Corpus and 2.05 bits/byte for the Canterbury Corpus. Some people prefer to split the Canterbury Corpus into two groups: the group of small files (alice29, ..., asyoulik) and the group of large files (the remaining). For the former group, we achieve an average compression rate of 2.14 bits/byte and, for the latter, it is 1.74 bits/byte. For each of the large files of the Canterbury Corpus, we could achieve even better compression rates by choosing a larger block size. (The results presented are for the block size of 900,000 characters.) The clear winner in this comparison is our program. There are other programs which achieve slightly better compression rates, but they require several orders of magnitude more compression and decompression time. Therefore, we excluded these from the comparison.

To demonstrate the practical relevance of our program, we measured its running time and compared it to *gzip*.

3. Note that, in our experiment, we have included the large files *e.coli*, *bible.txt*, and *world192.txt*, available at http://corpus.canterbury.ac.nz.

TABLE 3
Running Times (in Seconds) and Space (in Megabytes) for Calgary Corpus and Canterbury Corpus

		gzip		BK98		
file	*length*	*ctime*	*dtime*	*ctime*	*dtime*	*cspace*
bib	111261	0.35	0.04	0.87	0.32	1.16
book1	768771	3.64	0.25	9.63	2.80	8.32
book2	610856	2.14	0.18	6.56	2.01	6.52
geo	102400	1.00	0.06	2.26	0.58	0.87
news	377109	0.99	0.12	4.97	1.29	3.97
obj1	21504	0.06	0.02	0.20	0.11	0.19
obj2	246814	1.08	0.08	3.06	0.81	2.53
paper1	53161	0.14	0.03	0.41	0.18	0.57
paper2	82199	0.31	0.04	0.68	0.26	0.89
pic	513216	2.95	0.08	1.53	0.42	1.06
progc	39611	0.10	0.03	0.30	0.13	0.42
progl	71646	0.26	0.02	0.47	0.18	0.80
progp	49379	0.19	0.03	0.30	0.13	0.56
trans	93695	0.20	0.03	0.60	0.23	1.07
	3141622	13.40	1.01	31.85	9.45	28.93

		gzip		BK98	
file	*length*	*ctime*	*dtime*	*ctime*	*dtime*
alice29	152089	0.69	0.06	1.40	0.51
ptt5	513216	2.94	0.08	1.56	0.42
fields	11150	0.03	0.02	0.07	0.05
kennedy	1029744	36.36	0.23	39.45	3.12
sum	38240	0.42	0.03	0.34	0.14
lcet10	426754	1.66	0.13	4.39	1.36
plrabn12	481861	3.30	0.17	5.63	1.75
cp	24603	0.05	0.03	0.17	0.08
grammar	3721	0.02	0.01	0.04	0.03
xargs	4227	0.03	0.01	0.04	0.03
asyoulik	125179	0.50	0.05	1.18	0.43
ecoli	4638690	166.49	1.20	48.84	18.29
bible	4047392	25.65	1.00	39.33	12.09
world192	2473400	7.84	0.60	24.36	6.93
	13970266	245.97	3.62	166.81	45.23

Since *gzip* is available on almost every computer, these results allow a comparison to other programs. Table 3 shows compression time (*ctime*) and decompression time (*dtime*) for *gzip* and for BK98 when applied to the files of the Calgary and the Canterbury Corpus. It also shows the space our program requires for compressing the files of the Calgary Corpus (*cspace*). The last row gives the sums of the corresponding columns. The results were obtained on a computer with Pentium processor (166 MHz, 32 MB RAM) under the operating system Linux. We used the *gcc* compiler, version 2.7.2.3 with the optimizing option –O3. Times are user times in seconds (averaged over 10 runs) as reported by the *gnu time* utility. For the Calgary Corpus, *gzip* achieves about 2.4 times the speed of BK98 for compression. However, for the Canterbury Corpus, our program is about 1.5 times faster than *gzip*. We confirmed this surprising behavior on a different computer architecture: On a Sun-UltraSparc (143 MHz, 64 MB RAM), our program is 1.3 times faster than *gzip* when compressing the files of the Canterbury Corpus. For both corpora, *gzip* decompresses much faster than our program does. The space requirement for our program is on average about 9.5 bytes per input character when compressing the files of the Calgary Corpus. Similar results hold for the Canterbury Corpus. For lack of space, we cannot present them here in detail.

ACKNOWLEDGMENTS

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] M. Burrows and D. 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.

[2] F. Willems, Y. Shtarkov, and T. Tjalkens, "The Context-Tree Weighting Method: Basic Properties," *IEEE Trans. Information Theory,* vol. 41, pp. 653-664, 1995.

[3] Y. Shtarkov, "Universal Sequential Coding of Single Messages," *Problems Information Transmission,* vol. 23, no. 3, pp. 3-17, 1987.

[4] Y. Shtarkov, T. Tjalkens, and F. Willems, "Multialphabet Coding of Memoryless Sources," *Problems Information Transmission,* vol. 31, no. 2, pp. 20-35, 1995.

[5] T. Bell, J. Cleary, and I. Witten, *Text Compression.* Englewood Cliffs, N.J.: Prentice Hall, 1990.

[6] R. Arnold and T. Bell, "A Corpus for the Evaluation of Lossless Compression Algorithms," *Proc. Data Compression Conf.,* pp. 201-210, 1997. http://corpus.canterbury.ac.nz.

[7] J. Gailly, "The gzip Program, Version 1.2.4," 1993. ftp://prep.ai.mit.edu/pub/gnu/gzip-1.2.4.tar.gz.

[8] B. Balkenhol and S. Kurtz, "Universal Data Compression Based on the Burrows and Wheeler Transformation: Theory and Practice," technical report, Sonderforschungsbereich: Diskrete Strukturen in der Mathematik, Universität Bielefeld, 98-069, 1998. http://www.mathematik.uni-bielefeld.de/sfb343/preprints/.

[9] I. Witten, R. Neal, and J. Cleary, "Arithmetic Coding for Data Compression," *Comm. ACM,* vol. 30, no. 6, pp. 520-540, 1987.

[10] R. Krichevsky and V. Trofimov, "The Performance of Universal Encoding," *IEEE Trans. Information Theory,* vol. 27, pp. 199-207, 1981.

[11] J. Cleary, W. Teahan, and I. Witten, "Unbounded Length Contexts for PPM," *Proc. IEEE Data Compression Conf.,* pp. 52-61, 1995.

[12] P. Weiner, "Linear Pattern Matching Algorithms," *Proc. 14th IEEE Ann. Symp. Switching and Automata Theory,* pp. 1-11, 1973.

[13] E. McCreight, "A Space-Economical Suffix Tree Construction Algorithm," *J. ACM,* vol. 23, no. 2, pp. 262-272, 1976.

[14] E. Ukkonen, "On-Line Construction of Suffix-Trees," *Algorithmica,* vol. 14, no. 3, 1995.

[15] M. Farach, "Optimal Suffix Tree Construction with Large Alphabets," *Proc. 38th Ann. Symp. Foundations of Computer Science, FOCS 97,* 1997.

[16] U. Manber and E. Myers, "Suffix Arrays: A New Method for On-Line String Searches," *SIAM J. Computing,* vol. 22, no. 5, pp. 935-948, 1993.

[17] K. Sadakane, "A Fast Algorithm for Making Suffix Arrays and for Burrows-Wheeler Transformation," *Proc. IEEE Data Compression Conf.,* pp. 129-138, 1998.

[18] S. Kurtz, "Reducing the Space Requirement of Suffix Trees," *Software—Practice and Experience,* vol. 29, no. 13, pp. 1,149-1,171, 1999.

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

[20] R. Giegerich and S. Kurtz, "A Comparison of Imperative and Purely Functional Suffix Tree Constructions," *Science of Computer Programming,* vol. 25, nos. 2-3, pp. 187-218, 1995.

[21] R. Irving, "Suffix Binary Search Trees," research report, Dept. of Computer Science, Univ. of Glasgow, 1996. http://www.dcs.gla.ac.uk/rwi/papers/sbst.ps.

[22] M. Crochmore and R. Vérin, "Direct Construction of Compact Acyclic Word Graphs," *Proc. Ann. Symp. Combinatorial Pattern Matching (CPM '97),* pp. 116-129, 1997.

[23] N. Larsson, “The Context Trees of Block Sorting Compression,” *Proc. IEEE Data Compression Conf.,* pp. 189-198, 1998.
[24] B. Ryabko, “Data Compression by Means of a Book Stack,” *Problems Information Transmission,* vol. 16, no. 4, pp. 16-21, 1980.
[25] R. Ahlswede, T. Han, and K. Kobayashi, “Universal Coding of Integers and Unbounded Search Trees,” *IEEE Trans. Information Theory,* vol. 43, no. 2, pp. 669-682, 1997.
[26] Q. Stout, “Improved Prefix Encodings of Natural Numbers,” *IEEE Trans. Information Theory,* vol. 26, pp. 607-609, 1980.
[27] J. Rissanen, “A Universal Prior for Integers and Estimation by Minimum Description Length,” *Annals of Statistics,* vol. 11, pp. 416-431, 1983.
[28] V. Levenshtein, “On the Redundancy and Delay of Decodable Coding of Natural Numbers,” *Problems in Cybernetics,* vol. 20, pp. 173-179, 1968, (in Russian).
[29] P. Elias, “Universal Codword Sets and Representation of Integers,” *IEEE Trans. Information Theory,* vol. 21, pp. 194-203, 1975.
[30] P. Fenwick, “Block Sorting Text Compression—Final Report,” Technical Report 130, Dept. of Computer Science, Univ. of Auckland, 1996. http://www.cs.auckland.ac.nz/peter-f/ftplink/TechRep130.ps.
[31] G. Cormack and R. Horspool, “Data Compression Using Dynamic Markov Modelling,” *Computer J.,* vol. 30, pp. 541-550, 1987.
[32] A. Moffat, “Implementing the PPM Data Compression Scheme,” *IEEE Trans. Comm.,* vol. 28, no. 11, pp. 1,917-1,921, 1990.
[33] J. Seward, “The bzip2 Program, vers. 0.1pl2,” 1997. http://www.muraroa.demon.co.uk.
[34] M. Schindler, “The szip Homepage,” 1998. http://www.compressconsult.com/szip/.
[35] J. Ziv and A. Lempel, “Compression of Individual Sequences via Variable-Rate Encoding,” *IEEE Trans. Information Theory,* vol. 24, no. 5, pp. 530-536, 1978.
[36] J. Ziv and A. Lempel, “A Universal Algorithm for Sequential Data Compression,” *IEEE Trans. Information Theory,* vol. 23, no. 3, pp. 337-343, 1977.

Bernhard Balkenhol received the master’s degree in mathematics in 1992 and the PhD degree in mathematics in 1995, both from the University of Bielefeld, Germany. Since 1995, he has been an assistant professor in the Department of Mathematics at the University of Bielefeld. His current research interests include information theory, data compression, cryptography, and efficient algorithms for searching and sorting.

Stefan Kurtz received the master’s degree in computer science in 1990 from the University of Dortmund, Germany, and the PhD degree in computer science in 1995 from the University of Bielefeld, Germany. Since then he has been an assistant professor in the Department of Computer Science of the University of Bielefeld. In 1996/1997, he spent a year as a postdoctoral researcher at the University of Arizona. His current research interest is focused on bioinformatics, in particular on index structures for large biosequence databases and efficient algorithms for string processing.