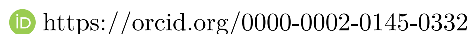
View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

On Undetected Redundancy in the Burrows-Wheeler Transform

Uwe Baier
Institute of Theoretical Computer Science, Ulm University
D-89069 Ulm, Germany
uwe.baier@uni-ulm.de

https://orcid.org/0000-0002-0145-0332

Abstract

The Burrows-Wheeler-Transform (BWT) is an invertible permutation of a text known to be highly compressible but also useful for sequence analysis, what makes the BWT highly attractive for lossless data compression. In this paper, we present a new technique to reduce the size of a BWT using its combinatorial properties, while keeping it invertible. The technique can be applied to any BWT-based compressor, and, as experiments show, is able to reduce the encoding size by $8 - 16\%$ on average and up to $33 - 57\%$ in the best cases (depending on the BWT-compressor used), making BWT-based compressors competitive or even superior to today's best lossless compressors.

2012 ACM Subject Classification Theory of computation → Data compression, Applied computing → Document analysis, Mathematics of computing → Coding theory

Keywords and phrases Lossless data compression, BWT, Tunneling

Digital Object Identifier 10.4230/LIPIcs.CPM.2018.3

Related Version Full version available at `https://arxiv.org/abs/1804.01937`.

Supplement Material Implementation available at `https://github.com/waYne1337/tbwt`.

1 Introduction

Lossless data compression plays an important role in modern digitization, as it enables us to shift and save computation resources during information exchange. For example, consider a setting where a computationally strong computer has to distribute data over a limited channel–by use of data compression, the storage requires less resources, and the file can be transmitted faster due to reduced data size–with the drawback of extra computation time for en- and decoding the information. Data compression is widespread today, current challenges are not only to compress data, but also to serve special features like resource-efficient decompression or even working on the compressed data directly, because the only way to fit it in memory (and thus process it fast) consists of using a compressed representation.

Compressors for the first mentioned feature typical make use of LZ77 [33]–a compression technique that, briefly speaking, replaces repeats in a text by references–resulting in very good compression rates and very fast decompression. Popular examples of compressors using LZ77 are `gzip` [12] or `7-zip` [27], which can be categorized as file transmission compressors.

A different technique for the second mentioned feature makes use of the Burrows-Wheeler-Transform (BWT) [3], which is an invertible permutation of the characters in the original text. The BWT itself does not compress data, but the transformed string tends to have some properties which make it highly compressible. The most popular compressor of such kind is `bzip2` [30], but compression rates are not the only aspect that make the BWT interesting:

© Uwe Baier;
licensed under Creative Commons License CC-BY
29th Annual Symposium on Combinatorial Pattern Matching (CPM 2018).
Editors: Gonzalo Navarro, David Sankoff, and Binhai Zhu; Article No. 3; pp. 3:1–3:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

also, the BWT in combination with wavelet trees [14] is well known to be an extremely useful and efficient tool for sequence analysis, commonly known as FM-index [8].

As the BWT has very interesting combinatorial properties, it has been heavily studied during the last two decades. For pure compression, a summary of the most relevant ideas can be found in [1]. Surprisingly, as Fenwick [6] stated, less attention was given to the encoding of runs (a run is a continuous sequence of the same character). This was reasonable, because, also stated by Fenwick [6], "the original scheme proposed by Wheeler is extremely efficient and unlikely to be much improved.".

Finally, this is the point where our paper draws on. In this paper, we describe a new technique called "tunneling", which relies on observations about combinatorics of a BWT and can be proved to maintain the invertibility (Theorem 13). The technique is independent from the way a run gets encoded, but reduces the size of the BWT to be encoded eminently by shortening runs. In numbers, we are able to reduce the encoding size of a compressed BWT by $8-16\%$ on average and up to $33-57\%$ in the best cases, depending on the backend used to compress a BWT. This not only makes BWT compressors competitive with today's best compressors, but also leaves the combinatorial properties of the BWT intact, what indicates that the technique is applicable to index structures such as the above mentioned FM-index[1].

This paper is organized as follows: Section 2 contains basics about the BWT and BWT compression. Section 3 presents our new technique, followed by a proof of invertibility of our representation in Section 4. Section 5 shows a way to implement the technique, which is followed by experimental results in Section 6 and conclusions in Section 7.

2 Preliminaries

First we want to describe the major parts of the BWT and its use in data compression. Throughout this Paper, any interval $[i, j]$ or $[i, j)$ is meant to be an interval over the natural numbers, every logarithm is of base 2, and indices start with 1, except when stated differently.

Let Σ be a totally ordered set (alphabet) of elements (characters). A string S of length n over alphabet Σ is a finite sequence of n characters originating from Σ. We call S nullterminated if it ends with the lowest ordered character $\$ \in \Sigma$ occurring only at the end of S. The empty string with length 0 is denoted by ε. Unless stated differently, we assume that S is nullterminated. Let S be a string of length n, and let $i, j \in [1, n]$. We denote by

- $S[i]$ the i-th character of S.
- $S[i..j]$ the substring of S starting at the i-th and ending at the j-th position. We state $S[i..j] = \varepsilon$ if $i > j$, and define $S[i..j) \coloneqq S[i..j-1]$.
- S_i the suffix of S starting at the i-th position, i.e. $S_i = S[i..n]$.
- $S_i <_{\mathsf{lex}} S_j$ if the suffix S_i is lexicographically smaller than S_j, i.e. there exists a $k \geq 0$ with $S[i..i+k) = S[j..j+k)$ and $S[i+k] < S[j+k]$.

▶ **Definition 1.** Let S be a string of length n. The *suffix array* [22] SA of S is a permutation of integers in range $[1, n]$ satisfying $S_{\mathsf{SA}[1]} <_{\mathsf{lex}} S_{\mathsf{SA}[2]} <_{\mathsf{lex}} \cdots <_{\mathsf{lex}} S_{\mathsf{SA}[n]}$.

▶ **Definition 2.** Let S be a string of length n, and SA be its corresponding suffix array. The Burrows-Wheeler-Transform (BWT) of S is a string L of length n defined as $\mathsf{L}[i] \coloneqq S[\mathsf{SA}[i]-1]$ if $\mathsf{SA}[i] > 1$ and $\mathsf{L}[i] \coloneqq \$$ if $\mathsf{SA}[i] = 1$. Also, we define the F-Column F as a string of length n by $\mathsf{F}[i] \coloneqq S[\mathsf{SA}[i]]$, which can also be obtained by sorting the characters in L.

[1] We give some hints towards this goal; there exist however more technical problems to be solved, outreaching the scope of this paper.

i	SA[i]	$S[1..\mathsf{SA}[i])$	$S_{\mathsf{SA}[i]}$	L[i]	F[i]	rlencode(L)[i]
1	10	easypeasy	$	y	$	y
2	7	easype	asy$	e	a	e
3	2	e	asypeasy$	e	a	0
4	6	easyp	easy$	p	e	p
5	1	ε	easypeasy$	$	e	$
6	5	easy	peasy$	y	p	y
7	8	easypea	sy$	a	s	a
8	3	ea	sypeasy$	a	s	0
9	9	easypeas	y$	s	y	s
10	4	eas	ypeasy$	s	y	0

Figure 1 Suffix array, Burrows-Wheeler-Transformation L, F, run-length encoded BWT rlencode(L) and the prefixes preceding a suffix (third column) for $S =$ easypeasy$. The BWT is almost the same string as the concatenation of the last characters from prefixes preceding suffixes, except for the sentinel character. Also one can see that those prefixes often share more than one character with the prefixes standing next to them.

In other words, the Burrows-Wheeler Transform is the concatenation of characters which cyclically precede suffixes in the suffix array. An example of a suffix array and a BWT can be found in Figure 1. An important property of the BWT is its invertibility, i.e. it's possible to reconstruct the original string S solely from its BWT. Therefore, we use the notation $\mathsf{rank}_S(c, i)$ to denote the number of occurrences of character c in the string $S[1..i]$, $\mathsf{select}_S(c, i)$ to denote the position of the i-th occurrence of c in S and $\mathsf{C}_S[c]$ to denote the number of characters smaller than c in S, that is, $\mathsf{C}_S[c] := |\{ i \in [1, n] \mid S[i] < c \}|$.

▶ **Definition 3.** Let S be a string of length n, SA and L be its corresponding suffix array and BWT. The LF-mapping is a permutation of integers in range $[1, n]$ defined as follows:

$$\mathsf{LF}[i] := \mathsf{C}_\mathsf{L}[\mathsf{L}[i]] + \mathsf{rank}_\mathsf{L}(\mathsf{L}[i], i)$$

We write $\mathsf{LF}^x[i]$ for the x-fold application of LF, i.e. $\mathsf{LF}^x[i] := \underbrace{\mathsf{LF}[\mathsf{LF}[\cdots \mathsf{LF}[}_{x \text{ times}} i] \cdots]]$, and define $\mathsf{LF}^0[i] := i$.

The LF-mapping carries its name because it maps each character in L to its corresponding position in F. To put it differently, the LF-mapping induces a walk through the suffix array in reverse text order, which commonly is called a "backward step".

▶ **Lemma 4.** *Let S be a string of length n, SA and LF be its suffix array and LF-mapping. Then, $\mathsf{SA}[\mathsf{LF}[i]] = \mathsf{SA}[i] - 1$ holds for all $i \in [1, n]$ with $\mathsf{SA}[i] \neq 1$.*

Proof. See [3]. ◀

Thus, any BWT can be inverted by computing LF (which can solely be done using L), taking a walk through the suffix array in reverse text order using LF and meanwhile collecting characters from L, what yields the reverse string of S (see [3] for more details). Our next concern of the LF-mapping which will be important later is its parallelism property inside runs, that is, a consecutive sequence of the same character in the BWT.

▶ **Lemma 5.** *Let S be a string of length n, L and LF be its corresponding BWT and LF-mapping. For any interval $[i, j] \subseteq [1, n]$ with $L[i] = L[i+1] = \cdots = L[j]$, $\mathsf{LF}[i] + k = \mathsf{LF}[i + k]$ holds for all $0 \leq k \leq j - i$.*

Proof. Follows directly from Definition 3. ◀

To get an understanding why the BWT is useful for data compression, we need a better understanding of it. The suffix array places lexicographically similar suffixes next to each other. Therefore, suffixes in subsequences of the suffix array often share a common prefix (context). As the BWT consists of the cyclic preceding characters of those suffixes, a subsequence of the BWT can be seen as a collection of characters preceding the same context in S. As a result, the character distribution inside a subsequence of the BWT gets skew, i.e. it is dominated by just a few characters which frequently appear before the context.

Typical BWT compressors make use of this fact by transforming the BWT such that the locally skew character distributions turn into a global skew character distribution–an example for such a transformation is given by the Move-To-Front Transformation [29, 3]. Finally, the global skew character distribution of the transform is useful for the last stage of typical BWT compressors: entropy encoding. The target of entropy encoding is to minimize the middle cost for the encoding of a character in a string using its character distribution. A well-known lower bound for this cost is given by the entropy definition of Shannon [31]:

▶ **Definition 6.** Let S be a string of length n, and let c_c be the count of character c in S. The entropy $H(S)$ of string S is defined as $H(S) := \sum_{c\in\Sigma} \frac{\mathsf{c}_c}{n} \log \frac{n}{\mathsf{c}_c} = \log n - \frac{1}{n} \sum_{c\in\Sigma} \mathsf{c}_c \log \mathsf{c}_c$.

Over the years, a couple of methods were developed to achieve cost-optimal entropy coding; most famous of such methods are Huffman coding [15] and Arithmetic coding [28]. However, it can be shown that the more skew a character distribution of an underlying source is, the more the entropy decreases. Consequently, the BWT transformation normally is highly compressible using an entropy encoder.

Another trick for improving compression used by most state-of-the-art BWT compressors is run-length-encoding. First of all, a run is a (length-maximal) subsequence in which all characters are equal. Run-length-encoding transforms a run with height (length) h of character c into the string $c\, h_k h_{k-1} \cdots h_1$, where $(1\, h_k h_{k-1} \cdots h_1)_2$ is the binary representation of h (in the encoding, the leading one is cut, and the symbols `0` and `1` are assumed to be distinct from symbols in S). Figure 1 shows an example of run-length-encoding which often reduces the length of a BWT drastically. We refer to [7] for a survey about further BWT compression methods, and introduce our last preliminary definition: the indicator function.

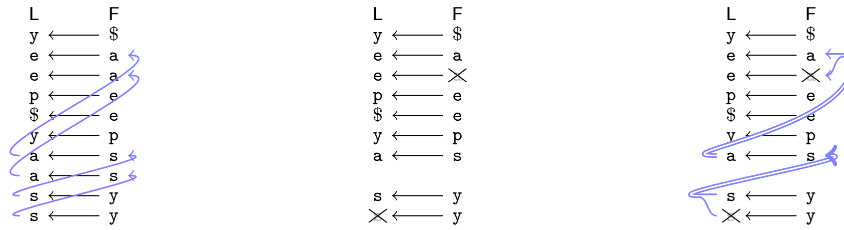
▶ **Definition 7.** Let P be any boolean predicate. The indicator function $\mathbf{1}_P$ then is defined as $\mathbf{1}_P := 1$ if predicate P is true, and $\mathbf{1}_P := 0$ otherwise.

3 Tunneling

The last section explained why the BWT produces long runs of the same character. Briefly speaking, the consecutive characters in the BWT are followed by similar contexts (suffixes) in the original text, and similar contexts tend to be preceded by the same character. The BWT limits the strings preceding contexts to just 1 character, but there is no reason why longer preceding strings shouldn't be similar too[2]. In fact, this often is the case in repetitive texts: In Figure 1, the suffixes $S_{\mathsf{SA}[9]}$ and $S_{\mathsf{SA}[10]}$ are both preceded by the same string `eas`.

The intention of this section is to show a way how to use the similarity of the preceding strings to reduce the size of the BWT while keeping the invertibility and combinatorial properties of the BWT intact. Unlike existing approaches [24], we will not use word substitutions to achieve this goal; the problem of word substitutions is to find a good substitution scheme,

[2] A similar observation was made in the context of self-repetitions in suffix arrays; see [21, 26].



(a) determine the block in the BWT

(b) cross out positions and remove doubly crossed-out ones

(c) Reconstruction idea: use one block row for both rows

Figure 2 Process of tunneling as described in Definition 9. Above, block $2 - [9, 10]$ from the running example is tunneled. Any lines colored blue are related to the LF-mapping, cross-outs in L and F are displayed by crosses.

as well as storing the word dictionary efficiently. Instead, we present a method based on the combinatorics of the BWT, which contains the "word dictionary" implicitly in the remaining BWT, and offers an easier way to find a good substitution scheme–completely without substitution. Our first step will be the definition of blocks, which are–short speaking–repetitions of the same preceding string in lexicographically consecutive suffixes.

▶ **Definition 8.** Let S be a string of length n and L be its BWT. A block B is a pair of an integer d and an interval $[i, j] \subseteq [1, n]$ ($d - [i, j]$-block) such that

$$\mathsf{L}\left[\mathsf{LF}^x[i]\right] = \mathsf{L}\left[\mathsf{LF}^x[i+1]\right] = \cdots = \mathsf{L}\left[\mathsf{LF}^x[j]\right] \text{ for all } 0 \leq x \leq d$$

We call $[i, j]$ the start interval of B, $\left[\mathsf{LF}^d[i], \mathsf{LF}^d[j]\right]$ the end interval of B, $h_B := j - i + 1$ the height of B and $w_B := d + 1$ the width of B.

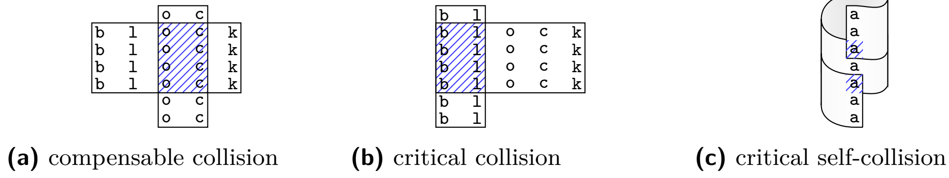
Blocks can also be seen as character matrices where each column consists of the same character and each row is build by picking characters in BWT during d consecutive applications of the LF-mapping, see also Figure 3. An example of blocks can be found in Figure 1: exemplary blocks are $2 - [9, 10]$, $1 - [7, 8]$, $0 - [2, 3]$ or $0 - [5, 5]$. We also want to note that each column of a block can be mapped to a substring in the BWT which consists of the same character.

▶ **Definition 9.** Let S be a string of length n, L and F be its corresponding BWT and F-Column, and let $B = d - [i, j]$ be a block of S. The process of tunneling block B is defined as follows:

1. cross out position $\mathsf{LF}^x[k]$ in L (mark it in a bitvector cntL of size n) for all $0 \leq x < d$ and $i < k \leq j$.
2. cross out position $\mathsf{LF}^x[k]$ in F (mark it in a bitvector cntF of size n) for all $0 < x \leq d$ and $i < k \leq j$.
3. remove positions k that were crossed out both in L and F from L, F and the bitvectors cntL and cntF.

To tunnel a whole set $\mathcal{B}$ of blocks, we apply each step to all blocks $B \in \mathcal{B}$ before continuing with the next step, where the result (L, cntL and cntF) is called a tunneled BWT.

A simpler description of tunneling a block can be given as follows: Remove all positions of the block from the BWT, except for those in the rightmost or leftmost column or uppermost row of the block. Afterwards, cross out positions in the interval start from L and the interval end from F, both except for the uppermost position. An example can be seen in Figure 2.



(a) compensable collision **(b)** critical collision **(c)** critical self-collision

Figure 3 Visualization of block collisions. Blocks are displayed as continuous stripes, shared positions are marked using diagonal lines.

The interesting question will be if we are able to reconstruct the text, even if we remove positions from the BWT. Beforehand however, we need to care about block intersections, since they can produce side effects when tunneling more than one block.

▶ **Definition 10.** Let $B = d - [i, j]$ and $\tilde{B} = \tilde{d} - [\tilde{i}, \tilde{j}]$ be two different blocks of a string S with BWT L and LF-mapping LF. We say that B and $\tilde{B}$ collide if they share positions in L, i.e. there exists $x \in [0, d]$ and $\tilde{x} \in [0, \tilde{d}]$ such that

$$\left[\mathsf{LF}^{x}[i], \mathsf{LF}^{x}[j]\right] \cap \left[\mathsf{LF}^{\tilde{x}}[\tilde{i}], \mathsf{LF}^{\tilde{x}}[\tilde{j}]\right] \neq \emptyset$$

We call each position in the intersection a shared position. Analogously, we say that a block $B = d - [i, j]$ is self-colliding if some $x, \tilde{x} \in [0, d]$ with $x < \tilde{x}$ exist such that

$$\left[\mathsf{LF}^{x}[i], \mathsf{LF}^{x}[j]\right] \cap \left[\mathsf{LF}^{\tilde{x}}[i], \mathsf{LF}^{\tilde{x}}[j]\right] \neq \emptyset$$

Furthermore, let B_{in} and B_{out} be two colliding blocks and w.l.o.g. $h_{B_{\mathsf{in}}} \geq h_{B_{\mathsf{out}}}$. We call the collision between B_{in} and B_{out} compensable if following conditions are fulfilled:

1. The leftmost and rightmost columns of B_{out} do not intersect:
   The positions $[i_{\mathsf{out}}, j_{\mathsf{out}}]$ and $\left[\mathsf{LF}^{d_{\mathsf{out}}}[i_{\mathsf{out}}], \mathsf{LF}^{d_{\mathsf{out}}}[j_{\mathsf{out}}]\right]$ are not shared
2. At least one row of B_{in} does not intersect:
   There exists a $y \in [i_{\mathsf{in}}, j_{\mathsf{in}}]$ such that the positions $y, \mathsf{LF}[y], \ldots, \mathsf{LF}^{d_{\mathsf{in}}}[y]$ are not shared
3. The intersection area forms a block of height $B_{\mathsf{in}} - B_{\mathsf{out}}$:
   For all $x_{\mathsf{in}} \in [0, d_{\mathsf{in}}]$ and $x_{\mathsf{out}} \in [0, d_{\mathsf{out}}]$ following holds:

$$\left| \left[\mathsf{LF}^{x_{\mathsf{in}}}[i_{\mathsf{in}}], \mathsf{LF}^{x_{\mathsf{in}}}[j_{\mathsf{in}}]\right] \setminus \left[\mathsf{LF}^{x_{\mathsf{out}}}[i_{\mathsf{out}}], \mathsf{LF}^{x_{\mathsf{out}}}[j_{\mathsf{out}}]\right] \right| \in \{0, h_{B_{\mathsf{in}}} - h_{B_{\mathsf{out}}}\}$$

If the conditions are not fulfilled, or if a collision is a self-collision, we call it critical.

Figure 3 visualizes the different kinds of block collisions. Examples of block collisions can be found in Figure 1: the blocks $2 - [9, 9]$ and $0 - [7, 8]$ form a compensable collision, while the blocks $2 - [9, 9]$ and $1 - [7, 8]$ form a critical collision. Furthermore, for $\mathsf{L} = \mathtt{aa} \cdots \mathtt{a\$}$ any block of width and height greater than one is self-colliding.

Now let us discuss the effect of the classifying criteria of collisions from Definition 10 a bit further. By the criteria, a compensable collision always consists of a "wider" (outer) and a "shorter but higher" (inner) block: The start and end interval of the outer block contain no shared position (condition 1), which in conjunction with condition 3 implies that the outer block must be wider than the inner one. For the inner block, at least one row must be unshared (condition 2), what in conjunction with condition 3 analogously shows that the inner block must be higher than the outer one. In the sense of visualization, these conditions build kind of a cross overlay of blocks, as depicted in Figure 3a. Extending this idea to more than two blocks, the criteria forms a natural hierarchy on colliding blocks, which will be useful for invertibility issues.

4 Invertibility

The purpose of this section is to show that a tunneled BWT can be inverted, i.e. the original string from which a BWT was constructed from can be rebuilt. As a first step, we will introduce a new generalized LF-mapping.

▶ **Definition 11.** Let L be the Burrows-Wheeler-Transformation of a string of length n and let cntL and cntF be two bitvectors of size n. The generalized LF-mapping is defined by

$$\mathsf{LF}^{\mathsf{cntL}}_{\mathsf{cntF}}[i] \;\coloneqq\; \underbrace{\mathsf{select}_{\mathsf{cntF}}(1\;,}_{\text{skip removed positions}}\; \underbrace{\sum_{k=1}^{n} \mathsf{cntL}[k]\cdot \mathbf{1}_{\mathsf{L}[k]<\mathsf{L}[i]}}_{\triangleq\; \mathsf{C}_{\mathsf{L}}[\mathsf{L}[i]]} \;+\; \underbrace{\sum_{k=1}^{i} \mathsf{cntL}[k]\cdot \mathbf{1}_{\mathsf{L}[k]=\mathsf{L}[i]}}_{\triangleq\; \mathsf{rank}_{\mathsf{L}}(\mathsf{L}[i],i)}\;)$$

Definition 11 is almost equal to the normal LF-mapping–except that characters crossed out in L are ignored, while characters crossed out in F are skipped. Next, we will see that this definition is reasonable for tunneling, as it maintains its structure when removing positions from L in the “right” manner.

▶ **Lemma 12.** *Let* L *be the Burrows-Wheeler-Transformation of a string of length n with LF-mapping* LF *and let* cntL *and* cntF *be two bitvectors of size n. Then following properties hold for the generalized LF-mapping $\mathsf{LF}^{\mathsf{cntL}}_{\mathsf{cntF}}$:*

1. *Let $\mathsf{cntF}[i] = \mathsf{cntL}[i] = 1$ for all $i \in [1, n]$. Then, $\mathsf{LF}^{\mathsf{cntL}}_{\mathsf{cntF}}$ is identical to the normal LF-mapping* LF.
2. *Let* cntF *and* cntL *be two bitvectors such that $\mathsf{LF}^{\mathsf{cntL}}_{\mathsf{cntF}}[j] = \mathsf{LF}[j]$ for all j with $\mathsf{cntL}[j] = 1$. Let i be an integer with $\mathsf{cntL}[i] = \mathsf{cntF}[\mathsf{LF}[i]] = 1$. Crossing out position i in* L *and position $\mathsf{LF}[i]$ in* F *(setting $\mathsf{cntL}[i] = \mathsf{cntF}[\mathsf{LF}[i]] = 0$) does not change the mapping: Define*

   $$\widetilde{\mathsf{cntL}}[j] \coloneqq \mathsf{cntL}[j]\cdot \mathbf{1}_{j\neq i} \quad \text{and} \quad \widetilde{\mathsf{cntF}}[j] \coloneqq \mathsf{cntF}[j]\cdot \mathbf{1}_{j\neq \mathsf{LF}[i]}$$

   Then, $\mathsf{LF}^{\widetilde{\mathsf{cntL}}}_{\widetilde{\mathsf{cntF}}}[j] = \mathsf{LF}[j]$ for all j with $\widetilde{\mathsf{cntL}}[j] = 1$.
3. *Let i be an integer with $\mathsf{cntL}[i] = \mathsf{cntF}[i] = 0$. Removing position i (crossed out both in* L *and* F*) from* L*,* cntL *and* cntF *does not change the mapping: Define*

   $$\widetilde{\mathsf{cntL}}[j] \coloneqq \mathsf{cntL}[j + \mathbf{1}_{j\geq i}]\;,\; \widetilde{\mathsf{cntF}}[j] \coloneqq \mathsf{cntF}[j + \mathbf{1}_{j\geq i}] \;\text{and}\; \widetilde{\mathsf{L}}[j] \coloneqq \mathsf{L}[j + \mathbf{1}_{j\geq i}]$$

   Then, for the corresponding mapping $\widetilde{\mathsf{LF}}^{\widetilde{\mathsf{cntL}}}_{\widetilde{\mathsf{cntF}}}$ the following holds:

   $$\widetilde{\mathsf{LF}}^{\widetilde{\mathsf{cntL}}}_{\widetilde{\mathsf{cntF}}}[j] = \mathsf{LF}^{\mathsf{cntL}}_{\mathsf{cntF}}[j + \mathbf{1}_{j\geq i}] - \mathbf{1}_{\mathsf{LF}^{\mathsf{cntL}}_{\mathsf{cntF}}[j+\mathbf{1}_{j\geq i}]\geq i}$$

Lemma 12 looks really bulky at the first moment, but reflects the operations required for tunneling, as we will see by discussing its different properties. Property 1 says that the new LF-mapping is identical to the old if we cross out nothing, and is straightforward. The more interesting property 2 tells us that the LF-mapping stays identical if we cross out consecutive positions in L and F in terms of text order. To explain why this works, think of a character c at position i in a BWT. If we cross out c in L, it is ignored, and thus the LF-mapping of all characters in L which are greater than c (or equal to c but placed below of i) gets shifted one position upwards, and thus is modified. Now, as we also cross out $\mathsf{LF}[i]$ in F, and crossed-out positions in F are skipped, all of the modified positions are shifted one position downward, because their original LF-mapping was greater than $\mathsf{LF}[i]$, and thus, the mapping stays identical. Property 3 also is easy to understand, as it says that a position which is

Algorithm 1: Inverting a tunneled Burrows-Wheeler-Transform.

Data: Tunneled Burrows-Wheeler-Transform L of size n from string S of size $\widetilde{n}$, bitvectors cntL and cntF of size $n+1$ with $\mathsf{cntL}[n+1] = \mathsf{cntF}[n+1] = 1$.
Result: String S from which L, cntL and cntF were build from.

```
1  let LF* be the generalized LF-mapping (Definition 11).
2  initialize an empty stack s
3  S[ñ] ← $
4  j ← 1
5  for i ← ñ − 1 to 1 do
6      if cntF[j + 1] = 0 then                          // end of a tunnel
7          j ← j + s.top()
8          s.pop()
9      S[i] ← L[j]
10     if cntL[j] = 0 or cntL[j + 1] = 0 then           // start of a tunnel
11         k ← max{l ∈ [1, j] | cntL[l] = 1}            // uppermost row of block
12         s.push(j - k)
13     j ← LF*[j]
```

ignored in L and skipped in F can be completely removed. A formal proof of the properties is omitted due to reasons of space, for now let us concentrate on the reconstruction of the original text.

The idea for reconstruction will be as follows: According to Lemma 12, tunneling will leave the LF-mapping identical, so we need only to clarify how to deal with tunneled blocks. As a remainder, tunneling means that we remove all of the characters except for the rightmost and leftmost column and uppermost row of a block. In blocks, we know that each row of the block is identical, and all of the rows run in parallel (in terms of the LF-mapping). Thus, if we reach the start of a tunnel, we will save the offset to the uppermost row, proceed at the uppermost row, and, once we leave the tunnel, use the saved offset to get back to the correct "lane", as shown Figure 2c and described in Algorithm 1.

The reconstruction process also inspired us to name the method tunneling: once the start of a block is reached, the offset to the uppermost row gets saved, and we enter the "tunnel", namely the uppermost row. After the tunnel ended, the temporarily information is used to get back to the correct "lane", that is the row on which we entered the tunnel.

▶ **Theorem 13.** *Let S be a string of length $\widetilde{n}$, let $\mathcal{B}$ be a set of blocks in its* BWT *containing no critical block collisions, and let* L*,* cntL *and* cntF *(each of size n) be the components emerging by tunneling the blocks of set $\mathcal{B}$. Then, Algorithm 1 reconstructs the string S from the tunneled* BWT *in $O(\widetilde{n})$ time.*

Proof. First, note that the generalized LF-mapping in a tunneled BWT conforms to the normal LF-mapping in a traditional BWT: Definition 9 tells us that for each marked position i in cntL, the associated position $\mathsf{LF}[i]$ is marked in cntF during tunneling process. Furthermore, tunneling only removes positions i with $\mathsf{cntL}[i] = \mathsf{cntF}[i] = 0$—note that this argumentation still is true for any colliding blocks, with only the only difference that some of those "position pairs" are marked more than once. Thus Lemma 12 ensures that a walk through the generalized LF-mapping (Line 13 of Algorithm 1) will reproduce the same string during character pickup (Line 9)—except for positions i with either $\mathsf{cntL}[i]$ or $\mathsf{cntF}[i]$ equal zero, which are positions in a start or end interval of a block.

As we know that each row of a block is identical (in terms of characters), and that the LF-mapping in a block runs in parallel (Lemma 5), for correct reconstruction, it's sufficient to store the relative offset to the top of a block when entering it, reconstruct any of the rows of the block, and at the end of the block use the stored offset to step back to the relative

position on which the block was entered—as performed by Algorithm 1. In case of block collisions, by the hierarchy build from the condition of compensable collisions, a tunnel will not be left until all its inner colliding block tunnels are left, thus the stack in Algorithm 1 correctly matches each tunnel end with the offset the tunnel was entered.

Finally, Algorithm 1 can be implemented to require $O(\widetilde{n})$ time by precomputing the generalized LF-mapping in $O(n)$ time, and by implementing line 11 with an array mapping each position to the nearest previous position i with $\mathsf{cntL}[i] = 1$, which obviously also can be computed in $O(n)$ time, requiring $O(1)$ time per query. ◀

In contrast, when dealing with critical collisions, Algorithm 1 will not be able to correctly match a tunnel start or end to the corresponding tunnel due to the intersections of start intervals or end intervals of blocks. In the case of self-collisions, the tunneling process from Definition 9 will remove entries of the topmost row of the self-colliding block from the BWT, thus falsifying the correctness proof of Algorithm 1.

5 Practical Implementation

This section's purpose is to give a brief summary on how to use BWT tunneling practically. Our first restriction therefore is to focus only on such-called run-blocks, what will make tunneling easier to handle. A run-block is a block whose start and end intervals have the same height as the runs in the BWT where the intervals occur. Furthermore, we will focus only on width-maximal run-blocks having height and width both greater than one.

▶ **Definition 14.** Let S be a string of length n and L be its BWT. Furthermore, assume that the border cases $\mathsf{L}[0]$ and $\mathsf{L}[n+1]$ contain characters such that $\mathsf{L}[0] \neq \mathsf{L}[1]$ and $\mathsf{L}[n] \neq \mathsf{L}[n+1]$.

A $d - [i, j]$-block is called a run-block if $\mathsf{L}[i] \neq \mathsf{L}[i-1]$, $\mathsf{L}[j] \neq \mathsf{L}[j+1]$, $\mathsf{L}[\mathsf{LF}^d[i]] \neq \mathsf{L}[\mathsf{LF}^d[i]-1]$ and $\mathsf{L}[\mathsf{LF}^d[j]] \neq \mathsf{L}[\mathsf{LF}^d[j]+1]$ holds.

A run-block RB is called width-maximal if it is wider than any colliding run-block $\widetilde{RB}$ with same height, i.e. RB and $\widetilde{RB}$ collide and $h_B = h_{\widetilde{RB}} \Rightarrow w_{RB} \geq w_{\widetilde{RB}}$.

An example of run-blocks is given in Figure 1: $0-[2,3]$, $1-[7,8]$ are run-blocks, $2-[9,10]$ is the only width-maximal run-block. As a counter example, $2-[9,9]$ is no run-block, as $\mathsf{L}[9] = \mathsf{L}[10]$, thus the height is not identical to that of the run where the block starts.

Run-blocks with height greater than one will never be self-colliding[3]–also, any collision between width-maximal run-blocks always is compensable: a run-block is height-maximal in sense of its start- and end-interval, thus any collision enforces one block to be higher and on the "inside" of the other, as required by Definition 10.

5.1 Block Computation

Our first concern is how blocks can be computed. For arbitrary blocks, a simple solution would be to compute the pairwise longest common suffixes of $S[1..\mathsf{SA}[i])$ and $S[1..SA[i+1])$, and afterwards enumerate the blocks using a stack-based approach–which is possible in $O(n)$, see [16] and [18]. However, as we want to compute the restricted set of width-maximal run-blocks, we will choose a different approach.

[3] Self-collisions are related to overlapping repeats in the text. Thus, a self-colliding block has to share positions in its start interval with itself, what cannot happen in run-blocks: as the overlapping intervals of a block cannot be equal (LF is a permutation), the start interval wouldn't be height-maximal.

L	cntL	cntF
y	1	1
e	1	1
e	1	0
p	1	1
$	1	1
y	1	1
a	1	1
s	1	1
s	0	1

(a) tunneled BWT from Figure 2

L	aux $= 2 \cdot$ cntL $+$ cntF
y	$3 = 2 \cdot 1 + 1$
e	$3 = 2 \cdot 1 + 1$
e	$2 = 2 \cdot 1 + 0$
p	$3 = 2 \cdot 1 + 1$
$	$3 = 2 \cdot 1 + 1$
y	$3 = 2 \cdot 1 + 1$
a	$3 = 2 \cdot 1 + 1$
s	$3 = 2 \cdot 1 + 1$
s	$1 = 2 \cdot 0 + 1$

(b) merge bitvectors cntL and cntL to vector aux

L	aux
→y	
→e	
e	2]
→p	
→$	
→y	
→a	
→s	
s	1]

(c) remove “run-heads” from aux and trim symbols beside of runs

Figure 4 Exemplary aux-vector generation from a tunneled BWT. “Run-heads” (first symbols of runs) are marked using arrows, runs with height greater 1 by right square brackets.

We will describe the idea of the approach only; see the full version of this paper for an algorithm. The idea is to use runs as a start point, and use the LF-mapping to proceed over the BWT. Every time a run is reached which allows to width-extend the current block, the current block is pushed on a stack and the run is used as new block. Then, as soon as the current block cannot be extended (because the current run is not high enough), blocks are popped from the stack until an extendable block is reached. During the removal of blocks, the necessary conditions for width-maximal run-blocks can be checked. Also, to increase efficiency, a side-effect similar to pointer jumping is used, which allows to skip already processed blocks.

5.2 Tunneled BWT Encoding

The encoding of a tunneled BWT requires to encode three components: the remaining BWT L, as well as the bitvectors cntL and cntF. To reduce a component, we merge cntL and cntF to a new vector named aux by setting aux$[i] := 2 \cdot$ cntL$[i] +$ cntF$[i]$. The vector aux now contains 3 distinct symbols[4], and is further shortened by removing all positions where runs in L start (all of this positions must be unmarked as the topmost row of a run-block stays unchanged) and by trimming the (identical) remaining symbols beside of each run of height greater than one to just one symbol (reconstruction is possible as only run-blocks are tunneled). An example is listed in Figure 4.

As the components L and aux originate from the same source and are quite similar, we’ll encode both components with the same BWT backend encoder, like, for example, Move-To-Front-Transformation + run-length-encoding of zeros + entropy encoding. This not only simplifies the implementation, but also allows to uncouple the block choice from the used backend encoder: As tunneling leaves the uppermost row of a block intact, the effect of tunneling can be summarized as shortening runs in L at cost of increasing the number of runs in aux due to the tunnel-marking symbols. For a good block-choice, it is useful to think of a tax system: a good choice maximizes the net-benefit, which is given by gross-benefit (amount of information removed from L) minus the tax (amount of information required to encode aux). Now, as long as different backend encoders encode runs in a similar fashion, an optimal block choice for a specific backend encoder will be near-optimal for all the other backend encoders, as the efficiency of such encoders can be seen as a constant c which does not affect the maximization of the net-benefit.

[4] Positions i containing markings in both cntL$[i]$ and cntF$[i]$ were removed by tunneling.

Algorithm 2: Greedy block choice

Data: a set $\mathcal{RB}$ of width-maximal run-blocks and a function `score` which for each block returns the amount of run-characters it removes.
Result: the array BS and a number t_{best}, whereby $\text{BS}[1..t_{\text{best}}]$ contains the blocks of a greedy block choice.

```
1  let BS be an array of size |RB|
2  t_best ← 0                              // number of tunnels allowing best benefit
3  b_best ← 0                              // best tunneling net-benefit in bits
4  tc ← 0                                  // number of run-characters removed by tunneling
5  for t ← 1 to |BS| do
6      let B ∈ RB be the block with score(B) maximal
7      reduce score(B̃) for all colliding blocks B̃ of B depending on the kind of collision
8      remove collisions between all inner and outer colliding blocks of B   // intersecting area gets
         tunneled
9      RB ← RB \ {B}
10     BS[t] ← B
11     tc ← tc + score(B)                  // update number of removed run-characters from tunneling
12     if gross-benefit − tax > b_best then   // update block choice; see equations (1) and (2)
13         t_best ← t
14         b_best ← gross-benefit − tax
```

As most state-of-the-art compressors use run-length-encoding, we estimate net-benefit and tax in terms of run-length-encoding. Consequently, the net-benefit between a normal BWT L and a tunneled BWT $\widetilde{\mathsf{L}}$ is given by

$$\begin{aligned}\texttt{gross-benefit} &:= |\texttt{rlencode}(\mathsf{L})| \cdot H(\texttt{rlencode}(\mathsf{L})) - |\texttt{rlencode}(\widetilde{\mathsf{L}})| \cdot H(\texttt{rlencode}(\widetilde{\mathsf{L}})) \\ &\approx n \log\left(\frac{n}{n-tc}\right) - rc \log\left(\frac{rc}{rc-tc}\right) + tc\left(1+\log\left(\frac{n-tc}{rc-tc}\right)\right)\end{aligned} \tag{1}$$

where $n := |\texttt{rlencode}(L)|$ is the number of characters of the run-length-encoding of L, H is the entropy function from Definition 6, rc is the number of run-characters in $\texttt{rlencode}(\mathsf{L})$ (all characters except for the run-heads) and tc is the number of removed run-characters in $\texttt{rlencode}(\widetilde{\mathsf{L}})$. The tax is given by

$$\begin{aligned}\texttt{tax} &:= |\texttt{rlencode}(\texttt{aux})| \cdot H(\texttt{rlencode}(\texttt{aux})) \\ &\approx 2 \cdot t \cdot (1 + \log(h^2 - 1)) + 2 \cdot t \cdot h \cdot \log\left(1 + \frac{2}{h-1}\right)\end{aligned} \tag{2}$$

where t is the number of tunneled Blocks and $h := \log\left(\frac{r_{h>1} - 2\cdot t}{2\cdot t}\right)$ with $r_{h>1}$ being the number of runs with height greater than 1.

5.3 Block Choice

The estimators from the last section give a clear indication that tunneling will not always improve compression–picking too small blocks may result in a tax whose size overcomes the gross-benefit. It is clear however that bigger blocks are preferable to smaller ones–thus a greedy approach will produce best results. Algorithm 2 sketches our strategy for choosing blocks, also considering that the gross-benefit and the tax do not grow in a likewise manner.

The collision handling of Algorithm 2 can be done using a collision graph, that is, a graph connecting colliding blocks whose intersecting area is not overlaid by a third block. Line 7 then can be implemented using a graph traversal together with block information, while Line 8 can be performed by removing the node corresponding to block B from the graph.

Implementing block scores is a bit complicated, as run-length-encoding is used. Initially, the score of a Block B gets to the sum of the run-lengths of all runs B points in minus the sum of all reduced run-lengths, i.e. the sum of $\log(h) - \log(h - h_B)$ for each run of height

h where B points in. Score updates of outer collisions are approximated by multiplying the score with the ratio by which the block width was shortened. Score updates of inner collisions can be done by subtracting the width-difference-ratio-multiplied score of B.[5]

By using a heap, Algorithm 2 can be shown to run in $O(n \log |\mathcal{RB}|)$ time. Unfortunately, we have to mention that the greedy strategy is not always optimal: think of three blocks whose colliding picture forms a shape like a big "H". If the middle block has a score bigger than that of the outer blocks (but close enough), for $t = 1$, the middle block is the optimal choice, while for $t = 2$ the outer blocks would be preferred, what is not covered by the algorithm. These situations however should not occur often in practice, so we neglected them.

6 Experimental Results

This section contains experimental results showing the effectiveness of our new technique. We applied our technique as described in the last section to three different BWT compressors:

- `bwz` : `bzip2` [30] without memory limitations.
- `bcm` : one of the strongest open-source-available BWT compressors [25, 19].
- `wt` : BWT encoded in a huffman-shaped wavelet tree[6] using hybrid bitvectors [11, 17] (good compression for miscellaneous data [17]) provided by the sdsl-lite library [13].

Our test data comes from three different text corpora, namely 12 medium-sized files from the Silesia Corpus [4] (6 – 49 MB), 6 bigger files from the Pizza & Chili Corpus [9] (54 – 1130 MB), as well as 9 repetitive files from the Repetitive Corpus [10] (45 – 446 MB). The full benchmark and all results are available at [2].

Beside of compression improvements, it is important to see if tunneling exploits its full potential not only in theory (as shown in the heuristics and approximations from last section) but also in practice. Therefore, we measured how well the theoretical model fits to the respective compressor by comparing the gross-net benefit ratios of model and compressor: as the efficiency of a compressor can be seen as some constant which is canceled when dividing two benefits from the same source, the gross-net benefit ratio should mostly be independent from the efficiency of a compressor, making it nicely comparable. Figure 5 shows compression improvements and the model fit as min-max distance $\frac{\min\{x,y\}}{\max\{x,y\}}$ of both ratios.

As the box plots show, the compression improvements of tunneling are significant and the theoretical model fits quite well. However, for half of the test files the encoding size decrease lies below $4 - 11\%$–not very surprisingly, this half typically consists of small to medium-sized files where the normal BWT-based compressors work very good already. Compression improvements for the upper half of the data however are significant and range from $4 - 11\%$ encoding size decrease up to $33 - 57\%$ decrease. Also, it seems that the better the compressor works (in terms of compression rate), the better the model fits.

The outliers in the model fit can be ignored for two reasons: first, as the figure shows, tunneling never worsens compression by a serious amount; second, the potential of tunneling in all those files (fraction of net-benefit of theoretical model and the size of the `bwz` encoding)

---

[5] Let B_{in} be the inner colliding block of B. We set $\texttt{score}(B_{\mathsf{in}}) = \texttt{score}(B_{\mathsf{in}}) - \frac{w_{B_{\mathsf{in}}}}{w_B} \cdot \texttt{score}(B)$, which is motivated by the fact that $\log(h - h_B) - \log(h - h_B - (h_B - h_{B_{\mathsf{in}}})) = \log(h - h_B) - \log(h - h_{B_{\mathsf{in}}}) = (\log(h) - \log(h - h_{B_{\mathsf{in}}})) - (\log(h) - \log(h - h_B))$ for a single run in which B_{in} and B collide.

[6] Note that this data structure is **not capable** of text indexing. By permuting the bits in cntL according to the permutation induced by stably sorting column L, the generalized LF-mapping can be computed using $\mathsf{select}_{\mathsf{cntF}}(1, \mathsf{rank}_{\mathsf{cntL}}(1, \mathsf{C_L}[\mathsf{L}[i]] + \mathsf{rank_L}(\mathsf{L}[i], i)))$, allowing backward steps using wavelet trees. Unfortunately, more technical problems must be solved, outreaching the scope of this paper.

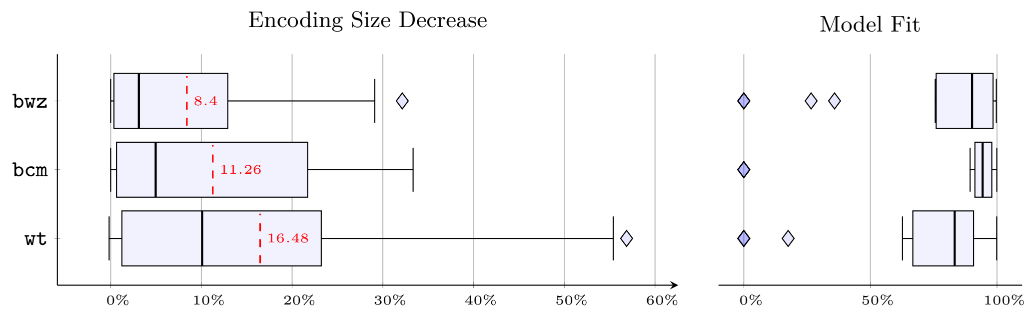


Figure 5 Compression improvements and model fit of tunneling displayed as Tukey boxplots. The boxplots contain the whole tested data set. Boxes consist of lower quantile, median, upper quartile and average (red dashed line), whisker range is given by 1.5 times the interquartile range, outliers are shown as diamond markers. Compression improvements use the untunneled versions as baseline, model fits are given by the min-max distance of the gross-net benefit ratios of theoretical model and compressor.

Table 1 Compression comparison of different compressors on a selection of the used test data. All values are shown in bits per symbol, that is, original file size times bits per symbol gives the compression size. Best compression results are marked bold.

Compressor	Silesia Corpus			Pizza & Chili Corpus			Repetitive Corpus		
	nci (32 MB)	samba (21 MB)	webster (40 MB)	proteins (1130 MB)	dna (386 MB)	english (1024 MB)	coreutils (196 MB)	para (410 MB)	world-leaders (45 MB)
`bwz`	0.34	1.81	1.48	2.29	1.83	1.84	0.23	0.31	0.12
`bwz -tunneled`	0.33	1.75	1.48	2.00	1.81	1.66	0.17	0.21	0.11
`bcm`	0.29	1.49	1.24	2.33	1.72	1.56	0.23	0.32	0.13
`bcm -tunneled`	**0.28**	1.42	1.24	**1.95**	**1.70**	**1.34**	0.16	0.21	0.11
`wt`	0.61	2.70	2.08	3.97	2.05	2.45	0.69	0.49	0.40
`wt -tunneled`	0.54	2.45	2.07	2.72	2.03	1.99	0.38	0.42	0.29
`xz -9e -M 100%` [32]	0.35	1.38	1.61	2.22	1.78	1.93	**0.14**	**0.11**	**0.09**
`zpaq isc` [20]	0.36	**1.20**	**1.21**	2.61	1.86	1.64	0.62	1.85	0.09

is below 0.3%. The model fit itself however shouldn't be overestimated, as we tested tunneling with different models, always getting a quite similar compression result.

Table 1 shows a comparison of our technique with compressors following different paradigms: `xz` [32] is a very effective Lempel-Ziv compressor similar to `7zip` [27], while `zpaq` [20] uses context-mixing. As the table shows, the tunneled version of `bcm` performs best among all shown BWT compressors, `xz` remains the best choice for repetitive data. Beside of pure compression, we want to note that tunneling has its price: the time and space requirements for encoding roughly double, while decoding time and space requirements are reduced by a small amount.

7 Conclusion

As we have seen in the last Section, compression gains due to our technique are a significant improvement to BWT-based compression. The technique however is in early stage of development, and therefore has some outstanding problems like making a good and economic block choice. A solution might be to use heuristics for similar problems in the LCP array

[5, 23], our presented approach is a nice baseline but too complicated and resource-expensive. It also would be nice to get rid of the restriction of run-based blocks; Section 5.1 indicates that this is possible, but collisions complicate the situation. In our opinion, the "big deal" will be to build a compressed FM-index [8] with full functionality; the footnote on page 12 clearly indicates that this should be possible, although correct pattern counting might be a bit tricky. Thinking of a text index with half of the size of the currently best implementations seems utopian, but this paper shows that this goal should be achievable, giving a lot of motivation for further research on the topic.

References

1 Jürgen Abel. Post BWT Stages of the Burrows-Wheeler Compression Algorithm. *Software – Practice and Experiences*, 40(9):751–777, 2010.

2 Uwe Baier. Tunneled BWT Implementation and Benchmark. `https://github.com/waYne1337/tbwt`. last visited January 2018.

3 Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

4 Sebastian Deorowicz. Silesia Corpus. `http://sun.aei.polsl.pl/~sdeor/index.php?page=silesia`. last visited January 2018.

5 Patrick Dinklage, Johannes Fischer, Dominik Köppl, Marvin Löbel, and Kunihiko Sadakane. Compression with the tudocomp Framework. In *16th International Symposium on Experimental Algorithms*, SEA '17, pages 13:1–13:22, 2017.

6 Peter M. Fenwick. Burrows-Wheeler compression: Principles and reflections. *Theoretical Computer Science*, 387(3):200–219, 2007.

7 Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The Engineering of a Compression Boosting Library: Theory vs Practice in BWT Compression. In *Algorithms – ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings*, ESA '06, pages 756–767, 2006.

8 Paolo Ferragina and Giovanni Manzini. Indexing Compressed Text. *Journal of the ACM*, 52(4):552–581, 2005.

9 Paolo Ferragina and Gonzalo Navarro. Pizza & Chili Corpus. `http://pizzachili.dcc.uchile.cl/texts.html`. last visited January 2018.

10 Paolo Ferragina and Gonzalo Navarro. Repetitive Corpus. `http://pizzachili.dcc.uchile.cl/repcorpus.html`. last visited January 2018.

11 Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When Indexing Equals Compression: Experiments with Compressing Suffix Arrays and Applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.

12 Jean-Loup Gailly and Mark Adler. `gzip` File Compressor. `http://www.gzip.org/`. last visited January 2018.

13 Simon Gog. `sdsl-lite` Library. `https://github.com/simongog/sdsl-lite`. last visited January 2018.

14 Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order Entropy-compressed Text Indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 841–850, 2003.

15 David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

16 Juha Kärkkainen, Dominik Kempa, and Simon J. Puglisi. Slashing the Time for BWT Inversion. In *Proceedings of the 2012 Data Compression Conference*, DCC '12, pages 99–108, 2012.

17 Juha Kärkkainen, Dominik Kempa, and Simon J. Puglisi. Hybrid Compression of Bitvectors for the FM-Index. In *Proceedings of the 2014 Data Compression Conference*, DCC '14, pages 302–311, 2014.

18 Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In *Proceedings of the 12th Annual Conference on Combinatorial Pattern Matching*, CPM '01, pages 181–192, 2001.

19 Matt Mahoney. Large Text Compression Benchmark. `http://mattmahoney.net/dc/text.html`. last visited January 2018.

20 Matt Mahoney. `zpaq` File Compressor. `http://mattmahoney.net/dc/zpaq.html`. last visited January 2018.

21 Veli Mäkinen. Compact Suffix Array. In *Proceedings of the 11th Annual Conference on Combinatorial Pattern Matching*, CPM '00, pages 305–319, 2000.

22 Udi Manber and Gene Myers. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, 5:935–948, 1993.

23 Markus Mauer, Timo Beller, and Enno Ohlebusch. A Lempel-Ziv-style Compression Method for Repetitive Texts. In *Proceedings of the Prague Stringology Conference 2017*, PSC '17, pages 96–107, 2017.

24 Alistair Moffat and R. Yugo Kartono Isal. Word-based Text Compression Using the Burrows-Wheeler Transform. *Information Processing and Management*, 41:1175–1192, 2005.

25 Ilya Muravyov. `bcm` File Compressor. `https://github.com/encode84/bcm`. last visited January 2018.

26 Gonzalo Navarro and Veli Mäkinen. Compressed Full-text Indexes. *ACM Computing Surveys*, 39(1), 2007.

27 Igor Pavlov. `7zip` File Compressor. `http://www.7-zip.org/`. last visited January 2018.

28 Jorma J. Rissanen and Glen G. Langdon. Arithmetic coding. *IBM Journal of Research and Development*, 23:149–162, 1979.

29 B. Ya Ryabko. Data compression by means of a “book stack”. *Problems of Information Transmission*, 16:265–269, 1980.

30 Julian Seward. `bzip2` File Compressor. `http://bzip.org/`. last visited January 2018.

31 Claude E. Shannon. A mathematical theory of communication. *The Bell System Technical Journal*, 27(3):379–423, 1948.

32 Tukaani. `xz` File Compressor. `https://tukaani.org/xz/`. last visited January 2018.

33 Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.